Department of Computer Science

The University of Waikato
Te Whare Wānanga o Waikato

Hamilton, NewZealand

Fast Algorithms for Nearest Neighbour Search

Ashraf Masood Kibriya

This thesis is submitted in partial fulfilment of the requirements for the degree of Master of Science at The University of Waikato.

March 2007

Abstract

The nearest neighbour problem is of practical significance in a number of fields. Often we are interested in finding an object near to a given query object. The problem is old, and a large number of solutions have been proposed for it in the literature. However, it remains the case that even the most popular of the techniques proposed for its solution have not been compared against each other. Also, many techniques, including the old and popular ones, can be implemented in a number of ways, and often the different implementations of a technique have not been thoroughly compared either.

This research presents a detailed investigation of different implementations of two popular nearest neighbour search data structures, KDTrees and Metric Trees, and compares the different implementations of each of the two structures against each other. The best implementations of these structures are then compared against each other and against two other techniques, Annulus Method and Cover Trees. Annulus Method is an old technique that was rediscovered during the research for this thesis. Cover Trees are one of the most novel and promising data structures for nearest neighbour search that have been proposed in the literature.

Acknowledgments

The continued support of Department of Computer Science's Machine Learning group, and particularly my supervisor Dr. Eibe Frank, is greatly appreciated, without which this thesis would not have been possible.

Contents

List of Figures

List of Tables

Chapter 1

Introduction: The Nearest Neighbour Problem

Nearest neighbour (NN) search is an old problem that is of practical importance in a number fields. It involves finding, from a given set of points, one or more points that are nearest to another given point, called the query point. The problem comes up in fields as diverse as data compression, computational biology, computer vision and information retrieval. It was originally proposed in 1969 by Minsky and Papert (Minsky & Papert, 1969). Since then, it has been extensively studied and a vast number of data structures, algorithms and techniques have been proposed for its solution. Though nearest neighbour 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.

1.1 Basic Definition

The NN search problem is:

> Given a set of n points S in some d-dimensional space X and a distance (or dissimilarity) measure M, our 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 which is nearest (or most similar) to q.

1.2 Extensions and Variations of the Problem

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

A slight variation of NN search, advocated by some (Indyk & Motwani, 1998; Datar et al., 2004) in place of NN search, is ϵ-approximate NN (ϵ-NN) search, where given a

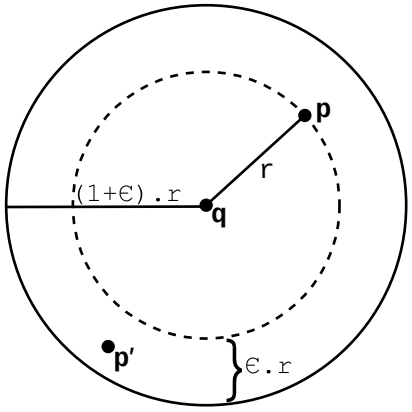


Figure 1.1: ϵ-Approximated Nearest Neighbour.

user defined error bound $\epsilon \geq 0$, the task is to find a point p' in S which is at most ϵ times farther than the exact NN p (also in S), i.e. $p, p' \in S \mid M(p', q) \leq (1 + \epsilon)M(p, q)$. For the kNN case this simply extends to finding k neighbours $p'_1, p'_2, ..p'_k$ such that any of the i^{th} ϵ-approximate NN p'_i is at most ϵ times farther than the exact i^{th} NN p_i, i.e. $M(p'_i, q) \leq (1 + \epsilon)M(p_i, q)$. Fig. 1.1 illustrates this graphically.

1.3 Applications of the NN Problem

The kNN search is of practical significance in a number of fields. Some of those, along with examples of their use of kNN search, include:

- **Data compression:** Here, it is used in a method called vector quantization for speech and image compression (Gersho & Gray, 1991). It involves blocking speech or image waveform signals into vectors of fixed length. A set of codevectors is first computed based on a set of training vectors, then each new vector is encoded with the index of its nearest neighbour among the codevectors.

- **Pattern recognition, datamining and machine learning:** Here, one of the most widely used classifier/learner is the kNN classifier (Cover & Hart, 1967). It is based on straight forward adoption of the kNN search, and works by assigning a given test point the majority class of its k-nearest neighbours. Also, Locally weighted learning (Atkeson et al., 1997) is another technique which utilizes kNN search. It works by training its base classifier/learner on training points that are nearest neighbours of a given test point.

- **Bioinformatics:** Here, kNN and its variant classifiers have been applied with success to biological and clinical data. They have been used for cancer classification

(Niijima & Kuhara, 2005), for detecting rRNA sequences (Robinson-Cox et al., 1995), and, using gene-expression data for tumour classification (Dudoit et al., 2002) and tissue classification, (Li et al., 2004). In cases involving gene selection (Niijima & Kuhara, 2005; Dudoit et al., 2002; Li et al., 2004), these classifiers have been observed to perform as well or even better than state-of-the-art SVM based classifiers.

- **Multimedia databases and libraries:** Here, similarity search is often used to retrieve multimedia content similar to a user query. The systems usually allow content-based queries, i.e. queries in the form of object shapes, texture, dominant colours, and scene descriptions etc. for images and video (Flickner et al., 1995; Pentland et al., 1994; Smith & Chang, 1996; Bach et al., 1996), and in the form of dominant frequency and pitch (which can also be given as an acoustic input from the user) etc. in case of an audio/music library (Ghias et al., 1995; McNab et al., 1997; Tseng, 1999; Uitdenbogerd & Zobel, 2002; Zhu et al., 2003).

- **Computer vision:** Here, NN search is an important tool used for the task of object classification, which involves finding similarities between images (Marshall, 2006). It has also been applied recently in a prototype of a robust multi-target tracking system, which tracks players in a hockey rink (Cai et al., 2006).

- **Document/information retrieval:** Here, NN search is often used method to retrieve and rank documents given a user query (Lucarella, 1988; Deerwester et al., 1990; Faloutsos & Oard, 1995).

1.4 Characteristics Common to NN Applications

In almost all of the above, the general representation of objects of interest (documents, images, etc.) including the queries is as vectors or points in some real d-dimensional space $X = R^d$, sometimes also called the *feature vector* representation; since each of the dimensions of the vectors correspond to some feature or attribute of the objects they represent. The closeness or (dis)similarity of the vectors (or points) is measured using a measure M which is usually a metric (a.k.a. distance function). However, both the object representation and the choice of similarity/nearness measure are usually dependent on the application domain. In string classification, for example, the objects are generally represented as string sequences rather than vectors (Aggarwal, 2002; Mollineda et al., 2003), even in cases when the employed distance measure (usually edit distance) is a

metric (Mollineda et al., 2003). Similarly, the Dynamic Time Warping (DTW) distance measure used in speech recognition (Vidal et al., 1988), and the NEM_r shape-distance measure employed in IBM's QBIC system (Flickner et al., 1995), as noted by Faragó et al. (Faragó et al., 1993) and Fagin and Stockmeyer (Fagin & Stockmeyer, 1998) respectively, are not exact metrics.

Nevertheless, almost all of the studies of (k)NN search that were reviewed as part of the research for this thesis, have concentrated on points/vectors in a real d-dimensional *metric* space. Because often it is easier to represent objects in terms of vectors, and for some domain specific measure to be mapped to a metric. Some of the techniques that have been proposed for the problem have even been devised for some particular metric (e.g. the initial version of LSH (Indyk & Motwani, 1998)). Many times even the NN-search problem itself has been defined with insistence on points being in a metric space (Arya et al., 1998; Maneewongvatana & Mount, 2002; Indyk & Motwani, 1998; Indyk, 1998, 2002)[1]. This, though, is pretty restrictive, given some of the earliest studies of the problem used more general *measures* instead of metrics (Friedman et al., 1977).

1.5 Objectives and Scope of the Thesis

It was observed during the review of the literature that even the most popular of the large number of techniques proposed since the initial inception of the problem, have not been thoroughly evaluated against each other. Also observed was the fact that there exists some considerable variation in how many of these techniques can be implemented. This also has not been thoroughly investigated either (either theoretically or empirically). When this research project was started it was intended to implement and empirically evaluate all the algorithms and techniques proposed in the literature. However, given the sheer volume of solutions proposed, compounded with the fact of different ways of implementing many of them and their different possible parameter settings, plus the fact that many solutions have been proposed for some specific similarity measure, and not to mention the fact that a number of those deal with some slight variation of the problem (e.g. ϵ-kNN) or the difficulties of gaps that exist between abstract theoretical descriptions to concrete implementations of the algorithms, that the initially intended goal was deemed unattainable. This was further exacerbated by the fact that a large number of techniques that exist have been designed for external memory (in order to reduce the I/O overhead in databases), and often cannot be fairly compared to those designed to work in main memory.

[1] Indyk and his group have corrected this in their latest publications (Andoni & Indyk, 2006; Shakhnarovich et al., 2006a)

Hence, to make the project more tractable, the goal of the research was narrowed down to include only the most popular of the old and the most promising of the novel techniques, that (a) work in main memory, (b) are applicable to vector/point representation, (c) use the Euclidean metric as distance measure, (d) have not been cross-evaluated, and (e) support exact NN search. This resulted in a detailed investigation of two popular data structures, KDTrees and Metric Trees, and their different proposed implementation methods were compared against each other. Later, the best implementations of the two techniques were evaluated against each other, and against one independently developed older technique, the Annulus Method, and one recently proposed–promising but not well evaluated–data structure, the Cover Trees.

1.6 Thesis Overview

The remainder of this thesis is structured as follows. The next chapter is devoted to solving the NN problem. It discusses the properties that would be required of an ideal solution, and the apparent difficulties of arriving at one. It also lists and describes many of the methods proposed so far, including the ones that were selected for evaluation, along with the reasons of their selection. The chapter that follows next (Chapter 3) delves more deeply into the details of each of the selected methods. Chapter 4 outlines the strategy used for evaluating these methods and presents the results. First, the various construction methods of the two popular algorithms, KDTrees and Metric Trees, are compared, and then the best of their construction methods are cross-evaluated against each other and against the rest of the selected methods. The thesis concludes with some comments and remarks in Chapter 5.

Chapter 2

Towards Solving the NN problem

Given n d-dimensional data points, a simple linear search (the brute force method) takes O(dn) time to find an exact NN of a given query point, while taking O(dn) space and only O(1) preprocessing time (since no preprocessing is performed). Hence, in order to do better than that, a data structure or an algorithm must try to find a NN in time sublinear in either d or n, or both d and n. Moreover, it should try to achieve that in space and preprocessing time that is linear or near linear in d and n. Since intuitively it's far simpler to eliminate points rather than trying to reduce the number dimensions looked at during a NN search, all known NN search techniques with the exception of two, PDS & TV-Tree, try to achieve a query time[1] sublinear in n.

In terms of n, ideal solutions exist with logarithmic query time, and near linear preprocessing and linear space requirements, for $d \leq 2$. For $d = 1$, a simple binary search on a sorted array gives O(log$n$) query time in the worst case, while requiring only O(n) space and O($n$log$n$) preprocessing time (which equates to time required for sorting the array in this case). For $d = 2$, O(log$n$) query time in the worst case, with linear space and near linear preprocessing time, is possible using methods based on Voronoi diagrams (Lee, 1982; Aurenhammer, 1991). However, for $d > 2$, no known solution exists that can *guarantee* a sublinear query time while still keeping the space complexity linear and the preprocessing time near linear. Still, for moderate values of d (≤ 10), algorithms exist that, though not in the worst case, but in practice (i.e. expected case) give sublinear query time, with linear space and near linear preprocessing. However, for higher values of d ($\geq$10) almost all of the known techniques are plagued with what is called the *curse-of-dimensionality*[2]. Roughly speaking what this means is that for points which are distributed somewhat uniformly, as their dimensionality increases, they become more and more sparse/scattered, and tend to lie more towards the boundaries of the space, making them almost equally distant from each other. This breaks down the idea of locality of

[1] Time to find (k)NNs of given a query q, here and elsewhere.

[2] Originally coined by Bellman in 1961 in "Adaptive Control Processes: A Guided Tour", Princeton University Press, 1961.

the points, since the local neighbourhoods become so large that they tend to comprise the entire point space. As a consequence most of the proposed algorithms, when searching for the nearest neighbour of a given query, degenerate to simple linear search, since (almost) all the data points have to be looked at, and often they take longer than the linear search itself due to their involved overheads. The curse was the main motivation behind algorithms for ϵ-NN search, and also the newer algorithms that are based on the concept of *intrinsic dimensionality* (defined below) of the data. A more meticulous coverage of the curse can be found in (Hastie et al., 2001), (Katayama & Satoh, 2001) and (Fayyad et al., 1996). Also, there is some debate regarding the usefulness of the NN problem at such high dimensions (Beyer et al., 1999; Hinneburg et al., 2000).

In practical problems it often turns out that the data points, even though being embedded in a high dimensional space, are not so widely scattered after all. Due to the dependencies among the dimensions (i.e co-relation among the attributes/features of the objects) they are usually clustered in some lower dimensional subspace. Indeed, many of the algorithms that should break down in such cases (e.g. KDTrees, Metric Trees etc.) sometimes do not and work better than expected. Such low-dimensional nature in high dimensional embeddings is called the *intrinsic dimensionality* of the points. Some recently proposed techniques such as Cover Trees and Navigating Nets try to exploit this peculiar nature common in real world data to circumvent the curse.

A number of dimensionality reduction techniques are also common in practice to mitigate the effect of the curse, even though the inter-point distances are not exactly preserved after the reduction. In the document/information retrieval field some sort of dimensionality reduction technique is essential, since it's often the case that $d \gg n$. Random Projections, Singular Value Decomposition (SVD), Principal Components Analysis (PCA), and Latent Semantic Indexing (LSI) (which is specific to IR), are some of the techniques that are generally used.

2.1 Techniques Proposed as Solution

Most of the techniques proposed for (k)NN and ϵ-(k)NN search, that are applicable to vector/point representation and try to achieve query time sublinear in n, are based around two general traditions of solving the problem. They either partition/decompose the point space to reduce the points looked at by eliminating regions unlikely to contain a NN, or they project the vectors to one or more scalar values and somehow try to reduce the points looked at using the scalar projections of data and query vectors. Those of such

Exact NN	*Based on Space Partitioning*	
	1. KDTrees	6. X-Trees
	2. Metrics Trees	7. M-Trees
	3. vp-Trees	8. SR-Trees
	4. Voronoi Diagrams	9. TV-Trees
	5. R-Trees	10. VA-Files
	Based on Scalar Projection	
	1. Orchard's Method	4. LAESA
	2. Annulus Method	5. LSH
	3. AESA	
	Based on Intrinsic Dimensionality	
	1. Cover Trees	
Approx NN	*Based on Space Partitioning*	
	1. BBF-Trees (KDTrees with Priority Queue)	
	2. BBD-Trees	
	3. Hybrid Sp-Trees	
	Based on Scalar Projection	
	1. LSH	
	Based on Intrinsic Dimensionality	
	1. Navigating Nets	

Table 2.1: Exact and approx. NN techniques

techniques that have captured the interest of other researchers, and the novel ones based on the idea of trying to exploit the intrinsic dimensionality of the data, are summarized in Table 2.1. The only known techniques that try to achieve query time sublinear in d are Partial Distance Search (PDS) (also known as Partial Distance Calculation) and TV-Trees. TV-Trees, however, try to reduce not only d but also n (hence are also mentioned in Table 2.1). These and the techniques in Table 2.1 are described in brief in the paragraphs below.

2.1.1 Partial Distance Search (PDS)

Originally proposed by Bei and Gray (Bei & Gray, 1985) it provides only moderate acceleration on its own. However, it is extremely simple and is general enough to be applied in conjunction with almost any other technique known for NN search.

The technique works by stopping the calculation of a point's distance to the query, if at any point during the calculation, the accumulated distance of the point becomes larger than the distance of the best (k^{th})NN, encountered among all the points so far looked at during the search. So, for example, for any Minkowsky-p (L_p) metric (e.g. Manhattan (L_1) metric and Euclidean (L_2) metric) of the form: $d(x,y)_p = \left(\sum_{i=1}^{d} |x_i - y_i|^p\right)^{1/p}$, it will skip the sum calculation for all dimensions $> i$, if at i the sum becomes larger than the overall sum of the best (k^{th})NN among all the points so far encountered.

The technique can be applied to any other dissimilarity measure that accumulates or adds values for different dimensions during calculation of the distance.

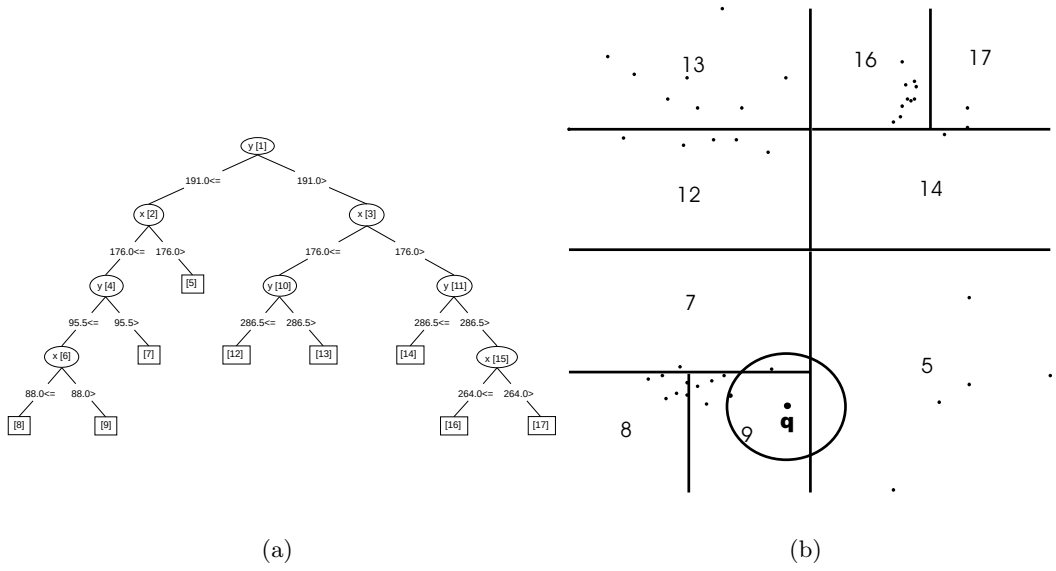
2.1.2 k-Dimensional Trees (KDTrees), BBF-Trees and Variants

Multidimensional binary search trees, called in short by the author as KDTrees (where k is the dimensionality of the space), were originally proposed by Bentley in 1975 (Bentley, 1975) for associative retrieval of records in a file. Their potential for NN search was observed by Bentley, and hence were quickly adopted for NN searching, with an optimized version by Friedman, Bentley and Finkel in 1977 (Friedman et al., 1977). Since then, these trees are by far the most popular search technique employed for NN search.

The trees hierarchically partition the point space into mutually exclusive rectangular regions by recursively splitting it with axis-parallel hyperplanes. The splitting is binary, with each non-terminal internal node splitting a region into two sub regions. The search for (k)NN is carried out recursively, with the region containing the query point being searched first and then only those of the remaining ones which are likely to contain the (k^{th})NN. More specifically, after recursively narrowing down to the region of a leaf node containing the query, the points inside the region are looked at, and then a ball (hypersphere to be exact), centred at the query and with radius equal to the query's distance to the best (k^{th})NN found so far, is computed. Afterwards during backtracking only those regions which intersect with this query ball are searched, and the ball is updated each time a better (k^{th}) NN is encountered in another region.

The trees require data in vector representation. They utilize this representation very efficiently and do not require any distance computation either during construction or during much of NN search (distance computations are performed only when looking at points inside a region of a leaf during the NN search). During both processes they only look at the value of a point's dimension that is orthogonal to the hyperplane used to split a region.

The original version proposed by Friedman, Bentley, and Finkel (Friedman et al., 1977) requires O(dnlog$n$) construction (preprocessing) time, and O(n) storage. For a given query it takes O(log$n$) time in the expected case for moderate dimensions. The query time, however, usually grows exponentially in d, and the tree, suffering from the curse-of-dimensionality, usually degenerates to simple linear search at higher dimensions (with slightly higher query time). The original version is general enough to be applied even to non-metric distance measures, and requires measures to satisfy only a few constraints given by the authors. Still, however, it is only known to be evaluated and found efficient



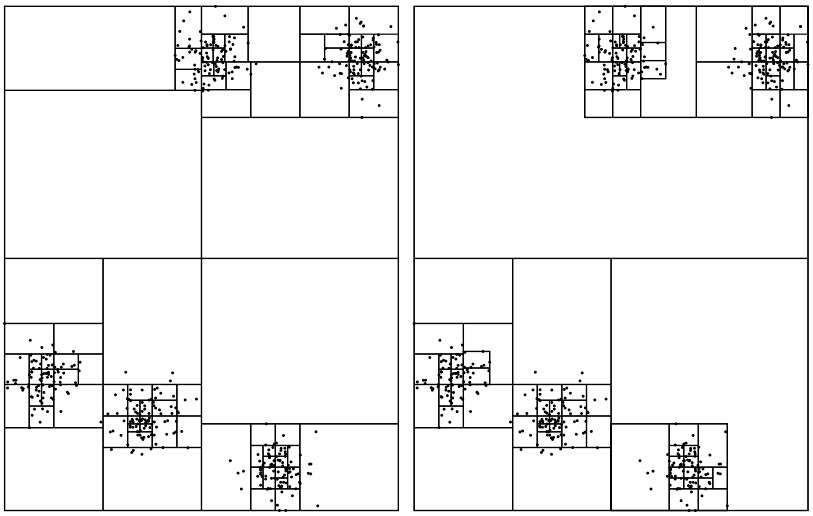
(a) (b)

Figure 2.1: (a) A typical 2 dimensional KDTree/BBFTree with (b) its graphical representation. The numbers in the rectangular regions in (b) correspond to the indices (given in brackets) of the leaf nodes in (a).

for Minkowsky-p metrics.

Since the initial version a number of different constructions methods, modifications, and enhancements have been proposed for the trees by different researchers. All, however, fall far short of removing the curse-of-dimensionality and all, from what is known, have only been devised with at most Minkowsky-p metrics in view.

Best-Bin-First (BBF) trees, developed by (Arya & Mount, 1993) and independently by (Beis & Lowe, 1997), are a modification of KDTrees for ϵ-NN search. Using a priority queue, they visit those regions first during back tracking that are nearer to the query point, and terminate the search early if the distance to the nearest region in the queue becomes greater than $1/(1+\epsilon)$ times the distance to the best encountered (k^{th})NN (i.e. $> 1/(1+\epsilon)\times$the radius of the query ball). As an example, consider Fig.2.1(b). For a given query q, after searching region 9, a typical KDTree would then search the leaf node of region 7, which is the first one encountered during backtracking as can be seen in Fig. 2.1(a). A BBF-Tree on the other hand, would first go on to search leaf 5 which is nearer to the query. If L' is a region associated with a node and r the radius of the query ball (equal to the distance of the best encountered (k^{th})NN), for a given distance measure M a KDTree would carry on the search until for all remaining nodes $M(q, L') > r$, whereas a BBF-Tree would terminate the search if for all remaining nodes $M(q, L') > 1/(1+\epsilon) \times r$. This way it can be guaranteed that any (k^{th}) neighbour found is at most by a factor of ϵ further than the true (k^{th}) nearest neighbour (see fig. 1.1).



(a) (b)

Figure 2.2: A (a) KDTree vs a (b) BBD-Tree on the same data.

A slight variation of BBF-Trees/KDTrees are the Balanced Box Decomposition (BBD) trees (Arya et al., 1998). The only difference between these and BBF-Trees/KDTrees is that for some nodes in BBD-Trees, their associated region is *shrunk* into two hyperrectangles (based on some shrinkage rule) instead of being split. The resulting two hyperrectangles after shrinking are set theoretic differences of each other and one of them completely encloses the other. These structures guarantee worst case ϵ-NN query time of $O(c_{d,\epsilon} \log n)$, where $c_{d,\epsilon} \leq d\lceil 1 - 6d/\epsilon \rceil$. However, some new enhancements to BBF/KDTrees by the same authors have removed the performance disparity between BBF/KDTrees and BBD-Trees, and the former are not known to perform significantly worse than the latter. The difference between the two trees is illustrated graphically in Fig 2.2.

2.1.3 Metric (Ball) Trees, vp-Trees and Hybrid Sp-Trees

Metric trees, which are also known as Ball Trees, were presented by Omohundro (Omohundro, 1989) and Uhlmann (Uhlmann, 1991a,b). These trees hierarchically partition the point space into (hyper)spherical regions. These regions, unlike the ones in KDTrees, are not mutually exclusive and are allowed to overlap. The points divided among the regions, however, are not allowed to overlap and can only belong to one region or one of its sub regions. Like in KDTrees, the partitioning is binary, and also like in KDTrees, the search for a (k)NN is done recursively, with the region containing (or in some cases nearest to) the query being examined first, followed by those regions that intersect with the query ball (i.e. the ball centred at the query, with radius equal to the query's distance to the

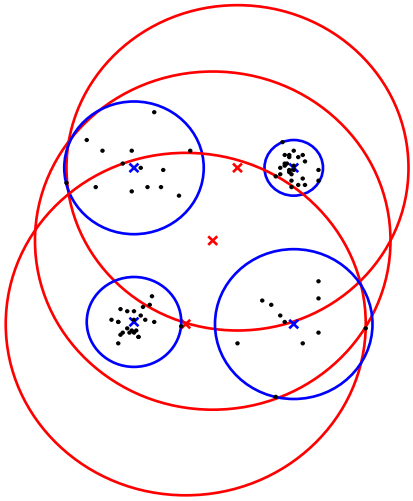


Figure 2.3: A typical metric tree decomposition.

best (k^{th})NN encountered so far, just like in KDTrees). Unlike KDTrees, they do not require data to be in vector form. They are general enough to be applied to any data representation as long as it is in a metric space.

The trees can be constructed in a number different of ways, given by a number of different people, with Omohundro (Omohundro, 1989) alone giving 5 different construction methods. Most of the proposed construction methods work in O($n$log$n$) time, while the space requirement regardless of the construction method, is always O(n). No bounds for query time are known to be given for any of the construction methods, and also no evaluation of the trees seems to have ever been carried out against established NN search methods such as KDTrees. In spite of that these trees have been the centre of attention of researchers recently (Moore, 2000; Liu et al., 2004; Liu et al., 2005), and are even claimed to be the state-of-the-art so far for exact NN search in moderately high dimensions (Liu et al., 2005); though without much theoretical or empirical underpinning. Like KDTrees, these trees are also known to suffer from the curse-of-dimensionality at higher dimensions. Fig 2.3 illustrates graphically a typical metric tree decomposition of 2-dimensional data.

Yianilos independently developed a slight variation of metric trees, which he called Vantage Point Trees (vp-Trees) (Yianilos, 1993). These trees divide the point space using spherical cuts into mutually exclusive (usually) semi-spherical regions. Each internal node is associated with a data point, called the vantage point, which is the centre of the spherical cut used to divide a (sub)space. The vantage point is usually chosen from one of the corner points in the (sub)space. Fig 2.4 graphically illustrates a vp-tree decomposition.

Liu et al. (Liu et al., 2005) noted that during (k)NN search of a given query a metric tree can spend up to 95% of the time in backtracking, i.e. after finding a good initial

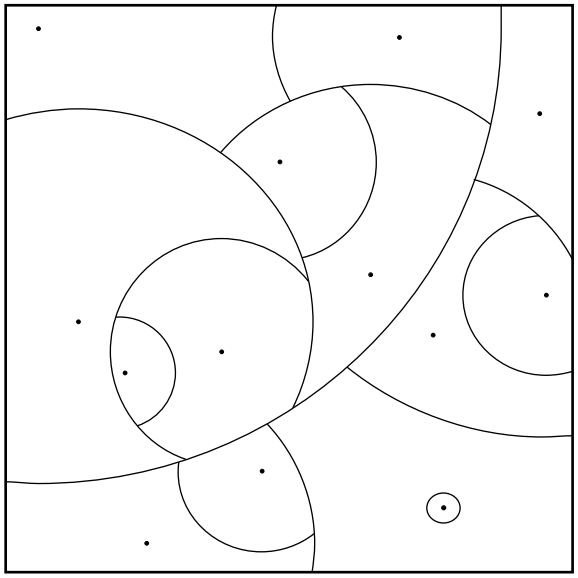


Figure 2.4: Illustration of a vp-tree decomposition.

candidate for the (k^{th})NN in a leaf node, up to 95% of its total time can be taken in making sure it has got the correct answer. To avoid this behaviour, they suggested a variation of metric trees that reports an approximate answer, that is an ϵ-(k)NN, but in a much shorter time. They called this variant Spill Tree (Sp-Tree). The key idea behind Sp-Tree is to allow data points that are at the split boundary to spill over between the two sub regions each time a region is divided, so that both sub regions contain the points near the boundary. The amount of spill-over is controlled by a distance parameter τ, which allows ϵ guarantees of ϵ-(k)NN. As noted by the authors, the Sp-Tree on its own is not pratically viable, as the distance parameter τ can introduce cases where a large number of points overlap and after division of a region both sub regions contain the same number of points as the parent region, thereby making the sub-division routine go into an infinite loop/recursion. Hence, for practical purposes they suggested a Hybrid Sp-Tree, a cross between Metric Trees and Spill Trees, that contains both overlapping and non-overlapping regions, and makes sure that at least a certain number of points of a region are divided among the sub regions. It marks a region overlapping only if after division with parameter τ both of its sub regions contain $< \rho$ of its total number of points (the authors set ρ to 70%), otherwise it is marked non-overlapping. The search for a (k)NN in a Hybrid Sp-Tree is also carried out in a hybrid fashion, with backtracking being skipped only for overlapping regions and not for non-overlapping ones. These structures work very well for ϵ-(k)NN, even in high dimensions. The authors in (Liu et al., 2005) also presented an empirical evaluation of the structures against the initial version of LSH (discussed below

in section 2.1.9), the only technique known to work well in high dimensions, and have shown Hybrid Sp-Trees to perform consistently better on a number of real-world datasets.

2.1.4 Voronoi Diagrams

Voronoi diagrams are of practical importance in a number of fields, which include, in addition to computer science, meteorology, physics, astronomy, archaeology and biology, e.g. see (Okabe et al., 2000). The diagrams have a long history of use, with their earliest use being traced back to 1644 when they were used by the well known French philosopher and mathematician René Descartes. Their general d-dimensional case was studied and formalized by Russian mathematician Georgy Fedoseevich Voronoy, whom they are named after. These are are also known as Voronoi tessellations or Dirichlet tessellations.

Voronoi diagrams partition a point space into convex polygons. The polygons are constructed in such a way that each polygon is associated with a single data point and any point that would be closer to this data point than any other would lie in this polygon. The NN search then just becomes a search for the region that contains the query point, and, because of the properties of Voronoi diagrams, the data point associated with that region is guaranteed to be the NN of the query. For kNN search with $k > 1$, an order-k Voronoi diagram of the data is computed. In order-k Voronoi diagrams, each polygonal partition of the point space is associated with exactly k data points, and any point falling in that region is closer to those k points than any other k points among the data. Figure 2.5 graphically illustrates an order-1 Voronoi space decomposition.

These structures are computationally only efficient for at most $d = 2$, for which they require O($n$log$n$) preprocessing and O(n) space. For values higher than 2, their complexity grows exponentially in d (O($n^{\lceil d/2 \rceil}$) according to (Arya et al., 2002)). The NN search on Voronoi diagrams takes only O(log$n$) time in the worst case, while kNN search takes only O(log$n$ + k) time. The search is done using an approach for planar point location in straight line graphs. For a long time, a major drawback of this technique was that k for kNN search needed to be known in advance before preprocessing, and had to remain fixed for all queries. However, an approach developed by Aggarwal et al. (Aggarwal et al., 1990) for compacting order-k Voronoi diagrams, allows all possible order-k Voronoi diagrams (i.e. for $k = 1...n - 1$) to be stored in O($n$log$n$) space, while still guaranteeing O(log$n$ + k) query time. An extensive survey of these structures is available in (Aurenhammer, 1991) and (Okabe et al., 2000).

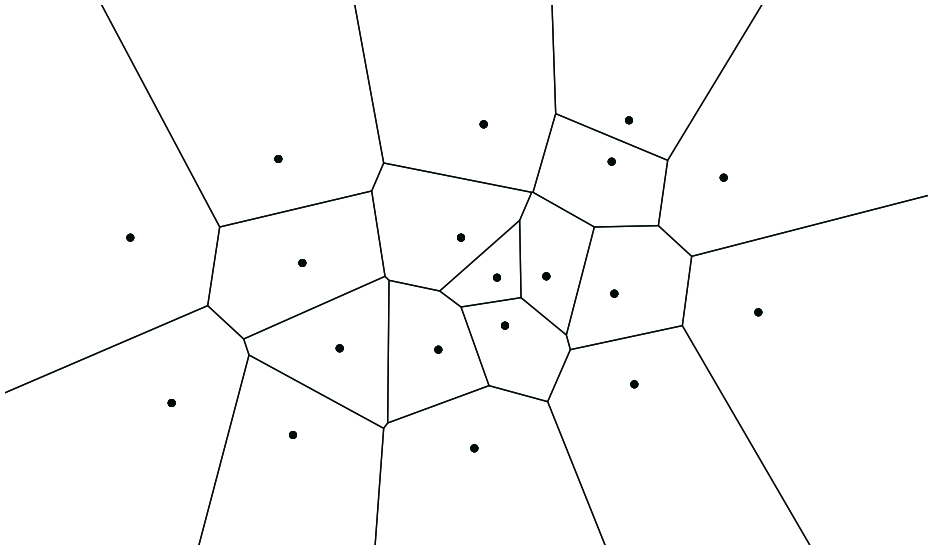


Figure 2.5: An order-1 Voronoi space decomposition.

2.1.5 Orchard's Method

Orchards's method (Orchard, 1991) was designed specifically for Vector Quantization. It works by selecting a point p as a candidate NN, computing its distance r to the query q, and then eliminating all the points that are further than $2r$ from p, thereby exploiting the geometrical property that they can not be nearer to q than p (See Fig. 2.6). After elimination, the method looks for a neighbour better than p, in the points remaining, within the circle of radius $2r$ centred at p (hypersphere in higher dimensions). If a neighbour better than p is found, the elimination procedure is applied again, and again a better neighbour is sought, in the new circle centred at the new best neighbour (with radius equal to twice the new neighbour's distance to the query). The selection of point p for the first query is arbitrary, and for subsequent queries, it turns out in image compression, that the best candidate is the NN of the last query processed, since a good initial candidate NN translates to faster performance.

The method requires for the elimination round computed (and sorted) distances from the point selected as a candidate NN to all the rest of the points. Since for the first and all the subsequent queries, any point can be selected as a candidate NN (which can subsequently be updated), it effectively requires all inter-data point distances to be computed at preprocessing, and thus requires $O(n^2)$ space and $O(n^2)$ pre-processing time; making it impractical for all but the smallest of NN problems. A slight variation of the method computes distances of all the points to only m other points, and requires only $O(mn)$ time and space. If during search a need arises to go past the m^{th} item on the sorted distance lists, it simply resorts to simple linear search. The original version (the

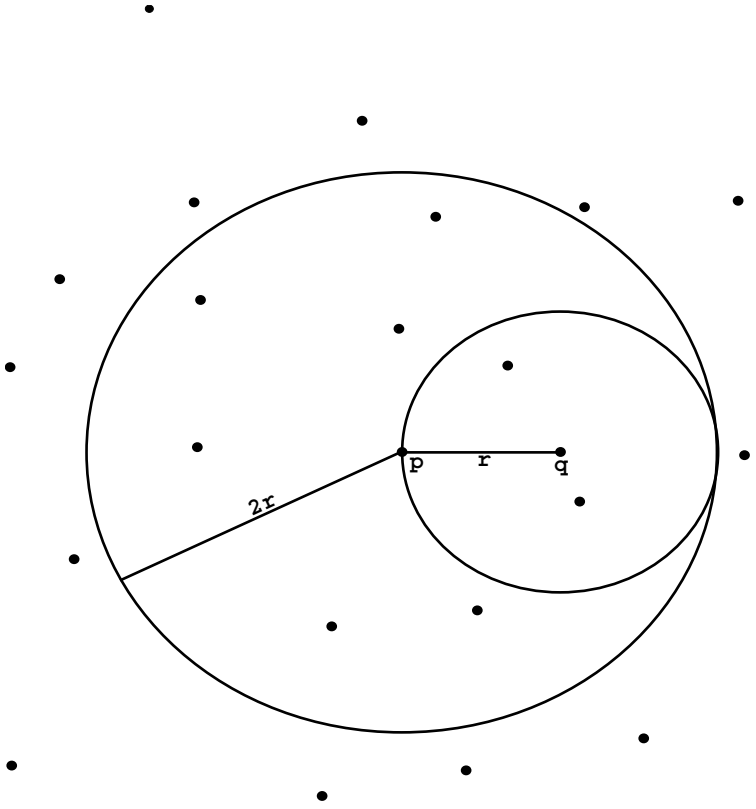


Figure 2.6: Orchard's method. All the points outside the circle of radius 2r are eliminated, and a better NN is sought in the remaining points inside the circle.

one with O(n^2) preprocessing) in terms of query time, has been demonstrated to work very well (Zatloukal et al., 2002). It compares favourably with KDTrees in moderate dimensions, and also works well in higher dimensions. However, for its variant that has O(mn) preprocessing, as the number of data points become large its performance degrades considerably, and even at higher dimensions it lingers behind KDTrees, which themselves are known to suffer from the curse-of-dimensionality.

2.1.6 Annulus Method

The Annulus Method, based around the mathematical concept of an *annulus* (a ring shaped object), was also designed specifically for Vector Quantization (Huang et al., 1992). Like Orchard's method, it also exploits a geometrical property. It works by projecting the points to their scalar distances from a fixed reference point (which is usually the origin). First, the distances of the data points are calculated, and they are sorted according to these distances and stored in an array or a list . Then, for a given query, its distance from the reference point is calculated and the data point with the distance closest to this distance is found in the array using binary search. The search for a (k)NN is then

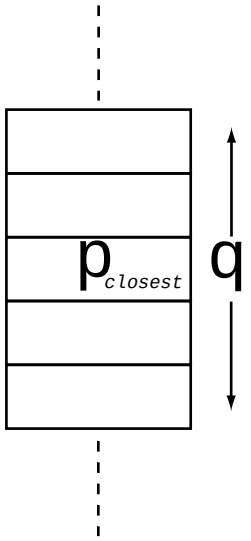


Figure 2.7: Annulus method for NN search. The point p's distance from the reference point is closest to that of the query q. The NN for the query is searched for in both directions (or only one, if p is the first or last element) in the array from the position of p.

carried out by looking at points in both directions in the array starting from the position of the point with the closest distance (see fig. 2.7). Geometrically, this corresponds to looking in the annular region around the reference point that contains the query (in higher dimensions, a hypertorus around the reference point), and as the search is carried out in both directions in the array, we effectively expand this annular region to include more points in space. The search stops when we encounter a point (or in the kNN case, the best k^{th} point) such that the circle centred at the query, with radius equal to the query's distance to this current point, is entirely within the annular region. The point found as such is geometrically guaranteed to be the (k^{th})NN. Fig 2.8 (a) and 2.8 (b) demonstrate this graphically. The number of points looked at is also geometrically guaranteed, and is bounded by the number of points in the annular region that contains the query and has width twice the query's distance to the true (k^{th})NN.

As is fairly evident, this method requires only O(n) space and O($n$log$n$) preprocessing time. The query time, as is intuitively conceivable, is demonstrated to grow linearly with the number of data points (Zatloukal et al., 2002), and as such the method is not as efficient as KDTrees for moderate dimensions. However, for higher dimensions, it does not seem to suffer from the curse-of-dimensionality, and, in the same study (Zatloukal et al., 2002), it is demonstrated to perform better than KDTrees.

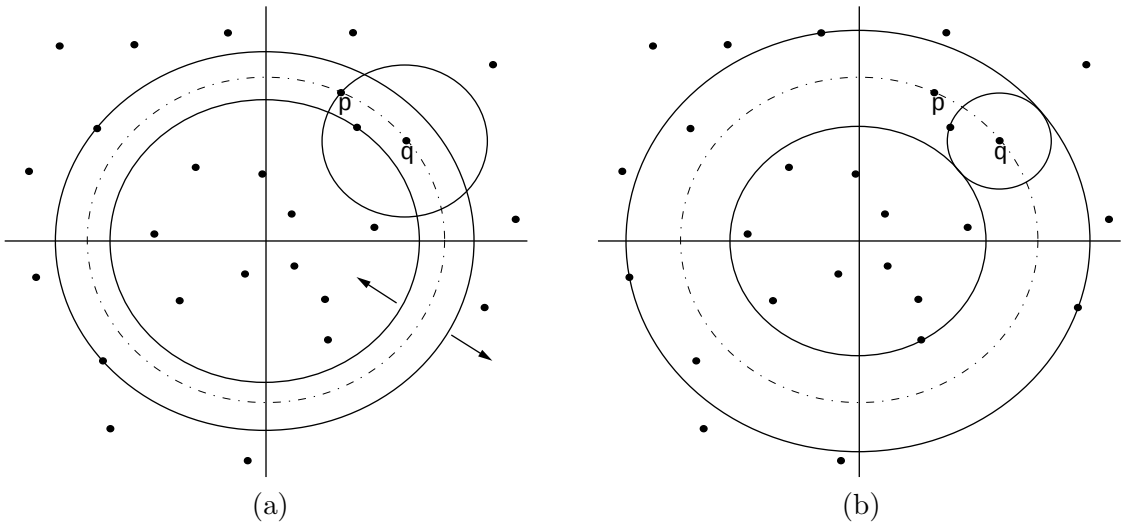


Figure 2.8: Graphical illustration of Annulus method.

2.1.7 AESA and LAESA

The approximating and eliminating search algorithm (AESA) was proposed by Enrique Vidal (Vidal, 1986). It is claimed to search on average only a constant number of points, implying that it has an expected query time of O(1). It employs the strategies of both Orchard's method and the Annulus method to eliminate points, and uses an approximation technique to select a point nearer to the query to eliminate more points until it finds the NN. The search starts by selecting an arbitrary point as a NN candidate and using Orchard's method's elimination criteria to eliminate all the points that cannot be nearer to the query than the selected candidate. The algorithm then goes into a loop of selecting the next candidate NN using an approximation technique (discussed below), and using the Annulus method's strategy to eliminate all those points that cannot be nearer to the query than the current best NN. In elimination, the algorithm looks at all the annular regions that are defined from all the points looked at so far as candidate NNs (apart from the current best NN). The search terminates when all the points have either been looked at or have been eliminated, and the best NN candidate found at the termination of the search is guaranteed to be the true exact NN. Figure 2.9 illustrates the elimination performed by the algorithm inside the loop.

To select the next candidate NN, the algorithm uses a numerical approximation to find a point nearest to the intersection of the circles that are centred at each of the points looked at (including the current best NN) and have radii equal to those points' distances to the query. The point of intersection of such circles is the query point itself, and approximating a point nearest to the intersection is therefore equivalent to approximating a point nearest

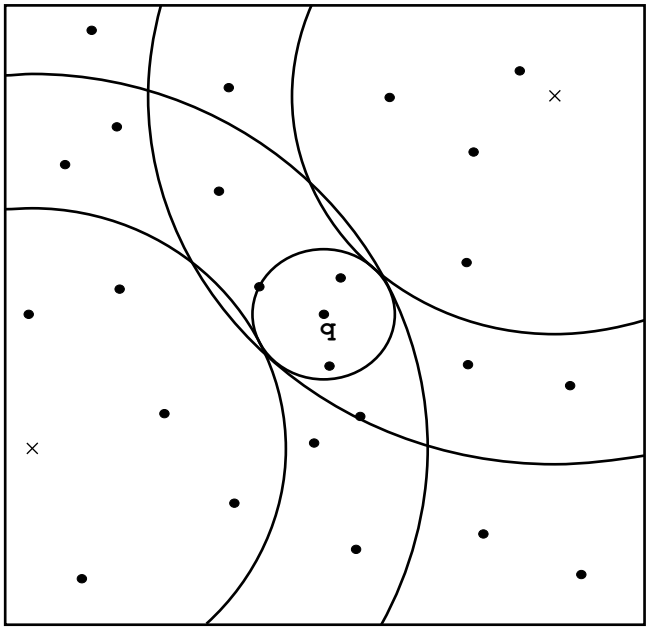


Figure 2.9: AESA's elimination. Points not falling in the annular region of all of the points so far looked at as candidate NNs (represented by x) are eliminated.

to the query. If the point found as such is not better than the current best NN (since it is an approximation), then it is simply added to list of points looked at. Figure 2.10 illustrates the technique graphically. In the figure the points represented as X are the ones that have so far been looked at, and the ones represented by circles are the ones that are being considered as candidates for the next NN. The circles centred at the points looked at are represented by arcs for clarity. The approximation chooses the point that has the minimum sum of lengths for its arrows to each of the circles centred at the points looked at, which in the case of the figure would be point $p3$.

For elimination, the algorithm requires distances from the points looked at, to all the remaining data points (for both Orchard's strategy at the start and Annulus' strategy during the loop), and hence requires $O(n^2)$ space and preprocessing time. A newer version developed by Micó, Oncina and Vidal (Micó et al., 1994), called the linear AESA (LAESA), removes this quadratic dependence of the algorithm, by selecting only a fixed set of points at the start of the search as elimination reference points. Only the points not lying in the annular regions of this set of points are removed, instead of annular regions of all the points so far looked at. The points selected as elimination reference points are the ones which are (approximately) at the corners of the point space. As shown by the authors, there is no single number of elimination reference points that can give best query-time performance in general. Instead, as was demonstrated by the authors, for best performance the number can be reasonably large even for moderate dimensions (check

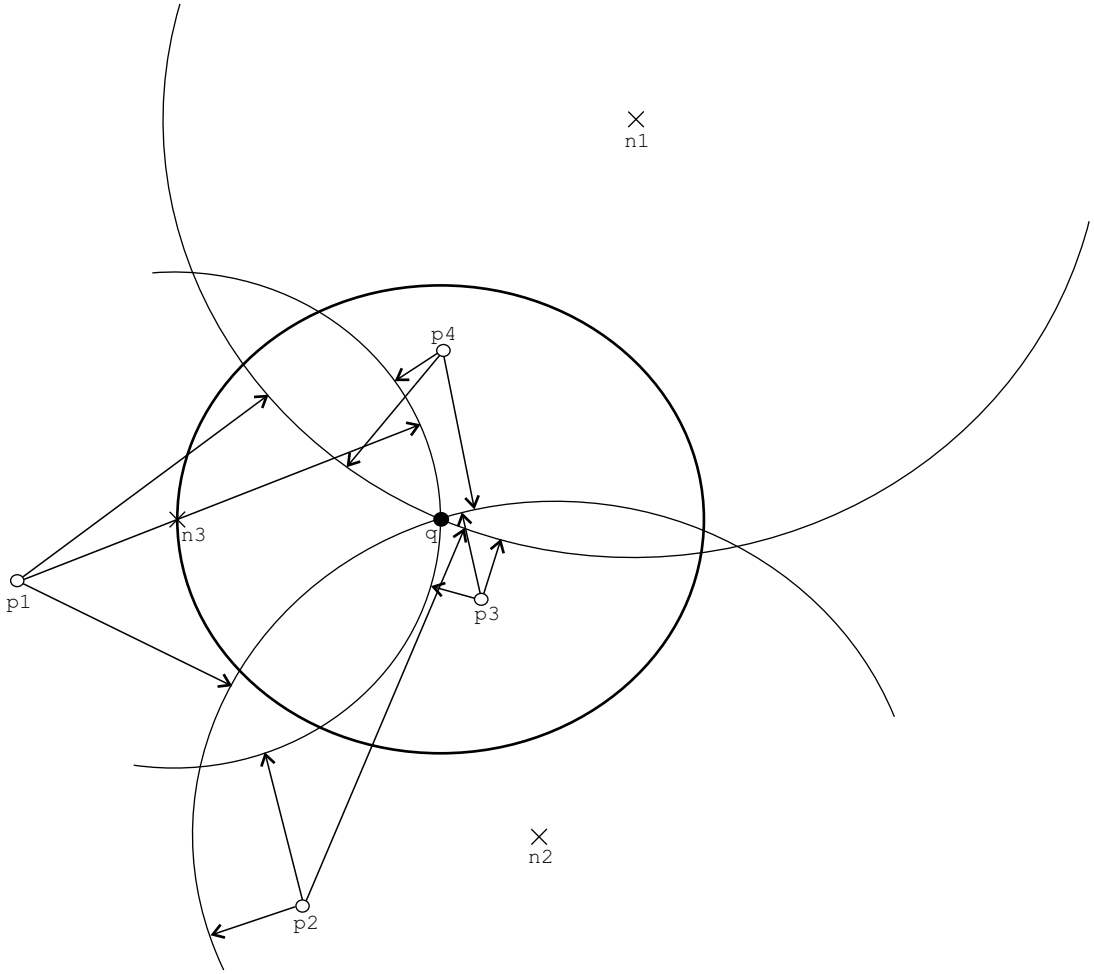


Figure 2.10: AESA's approximation technique. The point with the minimum sum of arrow lengths is selected (p3).

again in paper moderate or high), and hence incur a cost that can grow very large with increasing data size. In other words, it has a cn space and preprocessing requirement, which, with a large c, can become huge for larger n's.

2.1.8 R-Trees, X-Trees, M-Trees, TV-Trees, SR-Trees and VA-Files

These data structures are popular in the database community. They have been designed specifically to reduce the I/O cost associated with external memory, and to work well in a more dynamic setting (i.e. with efficient insertion and deletion operations since all the data points cannot be known in advance at construction).

R-Trees (Guttman, 1984), like KDTrees, hierarchically partition the data into hyperrectangles. However, the partitioning is achieved also using hyperrectangles instead of hyperplanes as in KDTrees. The partitioned rectangular regions are not mutually exclusive, and are allowed to overlap. The leaf nodes consist of the minimum bounding

rectangles of the points that they contain, and the internal nodes consist of minimum bounding rectangles of the points contained in their descendant leaves. A number of variants exist, that improve on the basic algorithm; such as the R+-Trees (Sellis et al., 1987), R*-Trees (Beckmann et al., 1990) and recently proposed Priority R-Trees (PR-Trees) (Arge et al., 2004). The variant R*-Tree is the most popular one used in practice.

X-Trees (Berchtold et al., 1996) are similar to R-Trees. The only major difference is that they employ an overlap-minimizing split algorithm and the concept of super nodes, which, as shown by the authors, enhances their performance by orders of magnitude compared to R*-trees and TV-Trees (discussed below).

M-Trees (Ciaccia et al., 1997), are the database variant of metric trees, with optimizations for reducing I/O costs. They hierarchically partition the point space, just like metric trees, into hyperspherical regions. The authors have demonstrated these to be comparable in performance to R*-trees. The key advantage of these structures, as noted by the authors, is their wider applicability, since, as opposed to R-Trees, they do not require objects to be represented as vectors, but only require them to be in a *metric* space. Concurrently but independently White and Jain (White & Jain, 1996) have also proposed similar structures, that are called SS-Trees, which also use bounding hyperspheres instead of bounding hyperrectangles for partitioning.

TV-Trees (Lin et al., 1994), like M-Trees, also use spherical bounding regions to partition points. In addition to that, they improve on R*-Trees by trying to reduce the effect of the curse-of-dimensionality by using only a subset of the dimensions to discriminate among the points. They rank all the dimensions by importance, and each sub-tree uses only one or more of the most important dimensions (called the active dimensions) for which the points have distinguishing values. If all the points have the same value for the most important active dimension for some subtree, then this dimension is deactivated and the most important dimension among the inactive ones is activated, thus *shifting* the active dimensions (called *telescoping*). Due to this telescoping (activating/deactivating) of the dimensions, the trees have a limited scope. They are effective only in problems where there is some sort of discreteness in the values of the dimensions, that would allow one or more objects to have the same value, and thus allow the idea of telescoping to work. The trees have been shown empirically by the authors to perform better than R*-Trees.

SR-Trees (Katayama & Satoh, 1997) are a combination of R-Trees and SS-Trees. Each node in the tree stores the minimum bounding rectangle as well as the minimum bounding sphere of the points it contains. On real data, the trees have been shown by the authors to perform better than both R*-Trees and SS-Trees. The trees have also been evaluated by

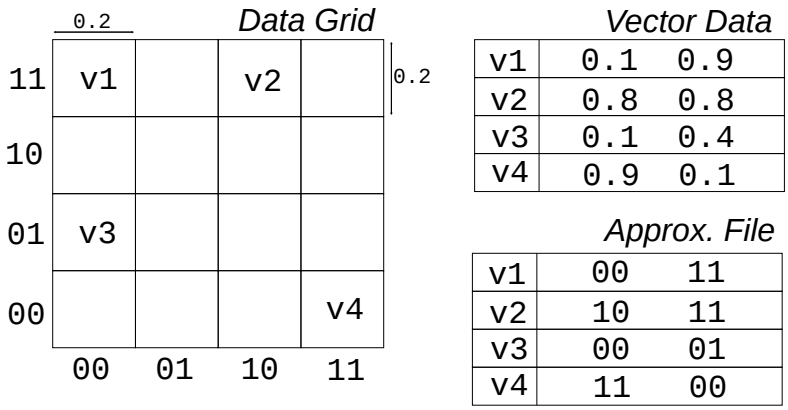


Figure 2.11: VA-File structure (Weber et al., 1998).

Liu et al. against Metric Trees, in which case, however, they perform consistently worse on a number of real-world datasets.

All of the above structures are known to suffer from the curse-of-dimensionality. Furthermore, Weber, Schek and Blott (Weber et al., 1998) have shown theoretically that, for a uniform hypercube, any techniques based on minimum bounding hyperrectangles or minimum bounding hyperspheres is likely to suffer from the curse. Thus, they proposed what they called *Vector Approximation file* (VA-File) structures (Weber et al., 1998; Weber & Blott, 1997). These structures divide the point space into a rectangular grid (not hierarchical) and use an approximation file to map each data point/vector to one of the cells in the grid using a fixed length identifier bit string. The unique bit string identifying each cell is itself a concatenation of individual bit strings identifying a small section of each of the dimensions. The approximation file used in the structures also stores, in addition to the bit strings of the data points, the lower and upper bounds of each dimension. Hence, given the number of sections along each dimension (which is 2^{b_i} for a bit string of length b_i for dimension i), the bounds of every cell along all the dimensions can be easily determined using the approximation file only. For a given query the whole approximation file is searched first, and lower and upper bound for the (k)NN of the query are determined, then only the vectors lying within these bounds are looked at in the data grid. The approximation file is much smaller compared to the data grid, to the tune of 12.5%-25%, and hence even with full examination of the file, considerable speed-up is achieved. The structures have been shown by the authors to outperform R*-Trees and X-trees. Fig. 2.11 illustrate a two dimensional VA-File structure, with the data grid, the approximation file and the corresponding data vectors.

2.1.9 Locality Sensitive Hashing (LSH)

Locality Sensitive Hashing (LSH) proposed by Indyk and Motwani (Indyk & Motwani, 1998), as the name implies, is based on a hashing scheme which is sensitive to position of the points. The technique was designed specifically for ϵ-NN search and especially to perform well in high-dimensions. However, it works only for Hamming metric space ($\{0,1\}^d$), and, for it to be utilized, objects represented in other metric spaces have to be mapped to Hamming space. The technique has a worst case query time of $O(dn^{1/(1+\epsilon)})$. It works well both in main and in external memory, and in a later study by the same authors has been shown to perform better than SR-Trees (Gionis et al., 1999). No evaluation of the technique, however, is known to have been carried out against established main memory methods such as KDTrees or BBF-Trees.

The basic idea behind the technique is to use a number of hash functions with the property that the probability of collision for points that are closer to each other is much higher than the probability for points that are further apart. First, during preprocessing, a certain number of hash functions are selected randomly from a family of hash functions which satisfy the said property (i.e. have higher probability of collisions for points closer to each other). Then each of the data points is hashed by each of the selected hash functions, and is then stored. At query time, a given query is also hashed by each of the used hash functions, and its (k)NN is then searched for in the buckets/cells returned by these functions.

Recently, a newer version of the technique has been proposed (Datar et al., 2004). It works for Minkowsky-p metrics for $p \in (0, 2]$. The new version, however, does not solve the ϵ-NN or the NN problem directly. Instead it solves a slight variation of the problems, called the near neighbour and its associated ϵ-near neighbour problem: given a query q and distance r, find a point within distance r from the query (or within $(1+\epsilon)\cdot r$ for ϵ-near neighbour). In case of (exact) near neighbour (which was never intended to be solved when the technique was proposed initially), there is no guarantee of sublinear worst-case query time (Shakhnarovich et al., 2006b), and it can only be achieved for datasets with *restricted growth* (explained in detail in the next section). The new version has been evaluated by the authors against BBF-Trees, and has been shown to perform better. Since, the new version does not deal directly with the ϵ-NN problem, its evaluation was carried out by ensuring the existence of a nearest neighbour within the specified distance r (by transformation in case of real datasets (Shakhnarovich et al., 2006c) and by way of generation in case of synthetic ones (Datar et al., 2004)). The authors, however, have not evaluated the new

version against Hybrid Sp-Trees, which are known to perform consistently better than the initial version of LSH. The performance of the new version compared to Hybrid Sp-Trees is thus an open question.

2.1.10 Navigating Nets and Cover Trees

These structures try to exploit the intrinsic dimensionality of a dataset (i.e. data points plus the query points). They work by placing assumptions that the datasets (or the metric spaces in which they are embedded), regardless of their actual number of dimensions, exhibit certain *restricted* or *bounded growth*. A simple notion of such bounded growth was presented by Karger and Ruhl (Karger & Ruhl, 2002). They defined a growth bound on a dataset such that the number of points in a ball (hypershere to be precise) centred at any point p is at most c times the number of points in a ball of half the radius centred at the same point; more formally, for all points p (in the dataset) and for all radii $r > 0$, $|B(p, 2r)| \leq c \cdot |B(p, r)|$. Their presented growth bound only allows points to come into view at a constant rate c (called the expansion rate), and rules out the possibility of suddenly encountering an exponentially high number of points as the ball around p is expanded. Such growth, as pointed out by Karger and Ruhl, occurs naturally in domains like peer-to-peer networks and the internet. Karger and Ruhl also presented a data structure for NN search which works well for geometries/datasets satisfying their growth bound.

A similar bound property was defined by Krauthgamer and Lee (Krauthgamer & Lee, 2004). Their growth bound, however, as shown in (Gupta et al., 2003), is more general than the one by Karger and Ruhl. Their growth bound definition is: every set of points in the dataset should be able to be completely covered with at most 2^{ρ} sets of half the diameter; or, in other words, any ball about a point p in the dataset should be able to be completely covered (in terms of the points it contains) with at most 2^{ρ} balls of half the radius. They defined the intrinsic or abstract dimensionality of a dataset, using this growth bound, as the minimum ρ for which this bound holds. This growth bound forms the basis of Navigating Nets, data structures for ϵ-NN search, which the authors also presented with their growth bound in (Krauthgamer & Lee, 2004). Navigating Nets work by arranging the points 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 $2^{i'}$ for 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. The structure is built in a greedy manner, where the first point in the list of points of the top level ball is used to build a smaller ball at the next level, and the first point inside the smaller ball is used to build a ball smaller at the level next. This is done recursively until we reach a level where a ball consists of a single point (on which that ball is centred). Then the build procedure back tracks to the last higher level cover ball that still has unprocessed points left, and picks the next point to greedily build cover balls at lower levels. Using the same terminology as Krauthgamer and Lee, if d_{max} is the maximum of the inter-point distances of the data points, d_{min} the minimum, and $\Delta = d_{max}/d_{min}$ the ratio between the maximum and the minimum inter-point distances, then the number of levels of a Navigating Net on a dataset is O(logΔ). If the dataset satisfies the growth bound given by authors, then every ball has at most O(1) cover balls at the lower level (because of the constant ρ).

The search for a (k)NN of a given query q is carried out by going down the levels of a Navigating Net and adding to a set of candidate NNs the children of a point whose ball intersects with the ball centred at the query. The radius of the query ball is set to the distance of the current best (k^{th})NN plus the radius of cover balls at the level currently being looked at. The search begins by adding the top-most point to the set of candidate NNs and setting the radius of the query ball as mentioned, so as to cover the entire point space, and then adding all the children of the top point to the set of candidate NNs. Then the search descends to the lower level, contracts the radius of the query ball (to the distance of the current best NN plus the radius of cover balls at this lower level) and adds the children of only those children of the top point whose balls intersect with the contracted query ball. The search carries on in this manner until we reach the bottom-most level or if at some level the current best (k^{th})NN cannot be at a distance more than ϵ farther from the query than the exact (k^{th})NN. Hence, at the end of the search the current best (k)NN is the ϵ-NN of the given query. For a dataset satisfying the authors' growth bound, the search procedure takes no longer than O(logΔ), and if the dataset instead satisfies the growth bound of Karger and Ruhl (which is a special case of Krauthgamer and Lee's growth bound), then the search procedure takes no longer than O(log$n$).

Beygelzimer, Kakade and Langford, presented a data structure for NN and ϵ-NN search based on Navigating Nets, which they called Cover Trees (Beygelzimer et al., 2006). In a Navigating Net each point at some lower level is allowed to have more than one parent point from the previous level (i.e. points are allowed to overlap among balls in any intermediate level), and also each level has also a pointer to the previous top level. In Cover Trees, the authors removed these redundancies to convert the graph rendered by a Navigating Net

into a tree, while still preserving the construction and query time. Furthermore, they also used the growth bound of Karger and Ruhl, as the one by Krauthgamer and Lee does not have strong theoretical guarantees for exact NN search.

Navigating Nets are not known to have been empirically evaluated against any other known technique for ϵ-NN search, whereas Cover Trees have only been evaluated (in the paper in which they were presented) against the little-known $sb(S)$ data structures by Clarkson (Clarkson, 1999, 2002). This is in spite of the fact that an excellent implementation of KDTrees and BBF-Trees, for NN and ϵ-NN search, is freely available (Mount & Arya, 1997) that can be easily integrated with any other data structure/technique for evaluation.

2.2 Techniques Selected for Evaluation

It can be noticed from the above brief survey of NN search techniques that most of them have not been compared (empirically) against each other, sometimes not even against the old and established ones. This is particularly true for main memory techniques not belonging to the database field. Even when the evaluation has been performed, and even for the techniques in the database field, often the conclusions have been drawn based on some particular domain of datasets, or sometimes even based on just one dataset only! Notable examples include (Gionis et al., 1999; Shakhnarovich et al., 2006c), who have only compared LSH to others on image data, (Zatloukal et al., 2002), who have only evaluated NN methods for image compression, and (Katayama & Satoh, 1997), who advocate their SR-Trees to be better than R*-Trees based on results for just one real-world dataset. In contrast Yianilos, when he evaluated his vp-Trees against KD-Trees (Yianilos, 1993), made sure that he stayed clear from drawing strong conclusions based on evaluation on a small sample of datasets that are never likely to be representative of all kinds of datasets found in practice.

For the research of this thesis, keeping in line with the objectives and scope outlined in Section 1.5, the following of the above techniques for exact NN search were selected for evaluation:

- **KDTrees**: They were selected because they are the oldest and one of the most established methods for NN search, and a number of construction methods have been proposed for them by different authors that have not been compared with each other. During the research that is presented in this thesis, another construction method was independently developed and was also investigated.

- **Metric Trees**: As mentioned above, they are claimed to be the state-of-the-art for moderately high dimensions (better than KDTrees). However, they are only known to have been properly evaluated against SR-Trees and never against the established KDTrees. Moreover, a number of different construction methods by different people were found in the literature that are also not known to have been empirically compared with each other.

- **Annulus Method**: A variant of the Annulus Method was independently developed during the research for this thesis. It was also compared against the rest of the techniques.

- **Cover Trees**: Cover Trees, as mentioned above, are the most novel technique proposed so far for NN search. However, as mentioned above, they have only been evaluated against one little known data structure, and hence were also selected for evaluation.

Orchard's method and AESA were not selected for evaluation because of their $O(n^2)$ construction cost, which makes them impractical even for modest data sizes. LAESA, the linear variant of AESA, was discovered very late during the research for this thesis and could not looked at. However, as noted by the authors, it also has a high space and construction cost, which even though constant makes the technique infeasible for large data sizes. Voronoi diagrams have been thoroughly studied, and are known to be the best solution for NN search for $d = 2$ (with linear space and near linear preprocessing), and hence did not warrant further attention in this thesis. LSH and Hybrid Sp-Trees, which are probably the only state-of-the-art for high dimensions (with linear space requirements), as mentioned above, deal with a slight variation of the NN problem, and hence fell outside the scope of this thesis. Same was the case for the techniques from the database field, which are not optimized for main memory and were not looked at. vp-Trees, the variant of Metric Trees discussed above, have been compared by their author to KDTrees (Yianilos, 1993), and were not found to be consistently better, and, also, they have not gained much popularity since they were first proposed (only (Brin, 1995) is known to have looked at them); therefore, they were also not looked at.

The evaluation of the selected techniques was carried out on a wide variety of synthetic datasets, many of which were generated so as to mimic the properties of the ones found in practice, while some were generated to try to maximize the worst-case behaviour of the selected techniques. This is discussed in Section 4.2.

Chapter 3

Evaluated Techniques in depth

The NN search techniques KDTrees, Metric Trees, Annulus Method, and Cover Trees, were selected for evaluation. These techniques are discussed in detail in the sections below. Throughout this chapter, and the remainder of the thesis, we'll assume points in a d dimensional Euclidean space X, and a set $S \subset X$ of n data points and a set $Q \subset X$ of m query points.

3.1 KDTrees

KDTrees are binary trees that partition/decompose the point space into hyperrectangular regions by hierarchically splitting it using axis-aligned hyperplanes. Each node of a KDTree is associated with a hyperrectangular region of the point space that it represents, with the root node being associated with a hyperrectangle comprising the entire point space. Internal nodes, in addition to the region they represent, also store information on the splitting hyperplane that splits their region, and also each of the two sub-regions as children that result from the split. Leaf nodes of the tree comprise of unsplit rectangular regions of the point space, and, using Friedman et al.'s (Friedman et al., 1977) terminology, are called buckets.

3.1.1 Basic Construction

A KDTree is constructed by recursively splitting the point space with axis aligned hyperplanes. Starting initially with a hyperrectangular region comprising the whole point space, a dimension and a value for that dimension, subsequently called *splitdim* and *splitval*, are chosen. Then, the points in the region with $splitdim \leq splitval$ (and their associated subregion) are assigned to left branch of the tree and the points with $splitdim > splitval$ are assigned to the right. This amounts to splitting the region with a $splitdim = splitval$ hyperplane that is orthogonal to *splitdim* and parallel to all other dimensions. The process is applied recursively to each of the sub-regions resulting from the split, and is carried

on until $\leq b$ points remain in a resulting sub-region. The number of points b (bucket size), at which the recursive splitting stops, is a user specified parameter. Figure 3.1 illustrates graphically the construction procedure on a 2-dimensional set of data points. It shows the first three steps of the construction process and the final result, on the tree and the point space. In the graphical representation of the tree, *splitdim* is shown inside the nodes, whereas *splitval* is shown on the edges. In the point space illustration, each splitting line is labelled by the number of the internal node in the tree figure that is associated with that line, where as the numbers inside the unsplit rectangular regions represent the associated leaf nodes in the tree figure.

The procedure above is for offline construction of the tree, where all the data points are available at the start of the construction method. Online construction of the tree is also possible with a point insertion procedure. In this procedure, for a given datapoint p, the leaf region (bucket) that contains this point is located by a procedure similar to the NN search described in the next section. Upon finding the bucket into which p falls, p is inserted into it, and if after insertion the number of points in the bucket becomes greater than the bucket size, then the bucket is split by choosing a suitable *splitdim* and *splitval*. This online construction method, however, is affected by the order in which the points are inserted, and the tree shape can differ widely for different orderings of the points. This insertion procedure can also be used in conjunction with the offline construction method, to insert points later on after the initial construction.

3.1.2 Basic query procedure

For a given query q, starting from the root node, the side of the splitting hyperplane containing the query is determined by comparing the query's *splitdim* value against the *splitval* stored in the node. This is done recursively until we find the region of the leaf node containing the query. On encountering the leaf region (bucket) containing the query, simple linear search is used to find and store the kNNs of the query from among the points of that region. Doing this is equivalent to computing the ball centred at the query with radius equal to the distance of the query to the best k^{th} NN encountered so far. During back tracking, the other side of the splitting hyperplane of each internal node encountered so far is inspected only if the query ball intersects with that hyperplane and overlaps with the region on the other side. If the query ball does overlap, then the same recursive search procedure is applied on the other side, and the query ball is shrunk each time a better k^{th} NN is encountered in another leaf region. Figure 3.2 gives an example of the NN search using this method. The arrowed line in the tree figure shows the nodes in the order they

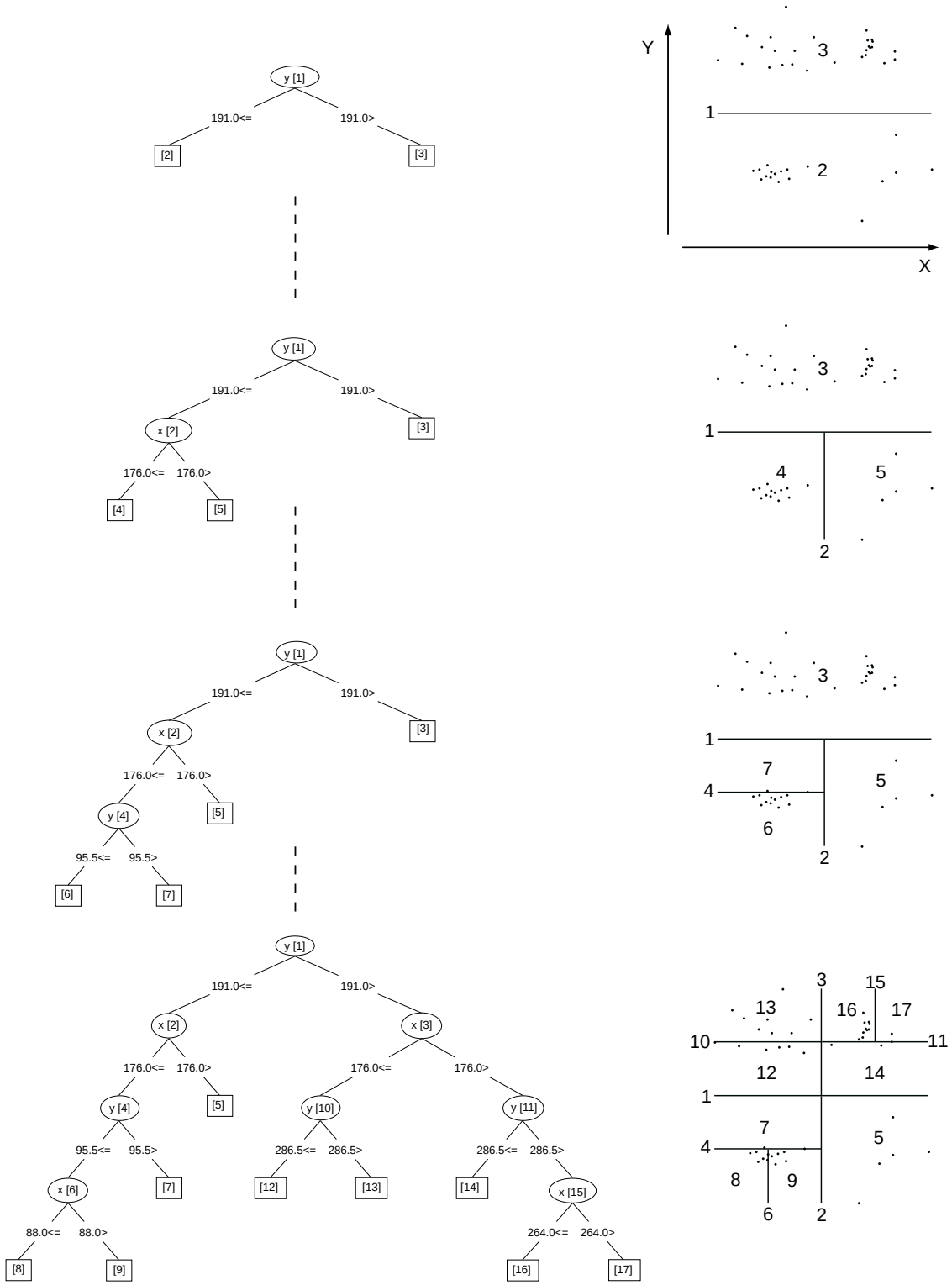


Figure 3.1: Illustration of KDTree construction. The first three steps and the final fully constructed tree.

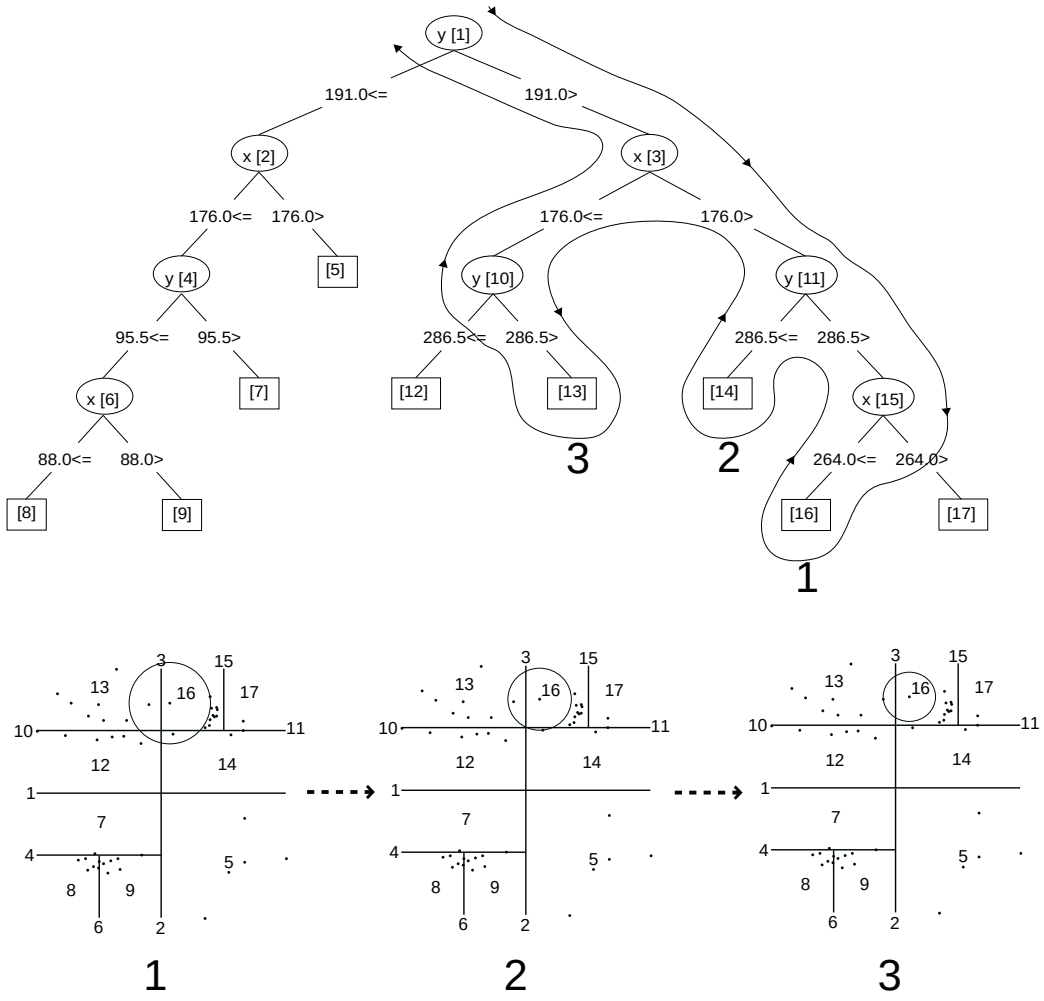


Figure 3.2: Illustration of KDTree query.

are visited, and the point space figure shows the query ball as it is updated each time a leaf region is visited.

3.1.3 Construction in detail

Let us now look at the construction procedure in more detail. The construction of a KDTree (both offline and online) is mainly affected by the choice of *splitdim* and *splitval*. Since the KDTrees were initially proposed a number of different suggestions have been made for the choice of *splitdim* and *splitval*. However, in most of the cases there does not exist much theoretical or empirical evidence on which one is the best choice in general.

During the review of the literature, it was observed that in the most recent past, most of the research on KDTrees has been carried out by of Andrew Moore's group, and group of David M. Mount and Sunil Arya. Mount and Arya have even provided a GNU licensed software library, called Approximate Nearest Neighbor (ANN) library (Mount &

Arya, 1997), that includes an excellent implementation KDTrees along with a number of construction methods, plus a number other NN techniques and a host of features for their evaluation. Moore and his group, in a number of their publications have either used the construction method of Friedman et. al (Friedman et al., 1977), which could be called the *Median of Widest Dimension*, or one their own proposed *Midpoint of Widest Dimension*. For example in (Moore, 1991), (Deng & Moore, 1995) and (Moore et al., 1997) they have used Midpoint of Widest Dimension, whereas in (Gray & Moore, 2004) they have suggested to use Median of Widest Dimension. Hence, it is not clear which method is best to use in general practice. Group of Mount and Arya have provided a number construction methods, and they have also provided theoretical and empirical studies comparing them to Friedman et al.'s Median of Widest Dimension. However, they have not looked at Midpoint of Widest Dimension method that is proposed by Moore and his group.

During the research of this thesis, another method for choosing *splitdim* and *splitval* was independently developed and tried out. It is based on inspiration from the K-Means clustering technique, and hence is called the *KMeans Inspired Method*. This method, and the one by Friedman et al., and the ones by groups of Moore, and Mount and Arya are described in detail below.

- **Median of Widest Dimension**: This was presented by Friedman et al. (Friedman et al., 1977). It is also called the standard method, or the standard splitting rule. This method chooses *splitdim* to be the dimension in which the points in a region to be split have maximum spread, and *splitval* to be the median point value in that dimension. Choosing *splitdim* and *splitval* as such guarantees a perfectly balanced tree of depth O(log$n$), with at most O(n) nodes, while taking O($n$log$n$) construction time. Also, the query time in expected case is O(log$n$). Uptil now, this original construction method is the most popular one and the one most widely used in practice.

- **Midpoint of Widest Dimension**: This method appears to have been first proposed by Moore in his Phd thesis (Moore, 1991) (most probably as no other study is known to have mentioned this method). This method, just like Median of Widest Dimension, chooses the *splitdim* to be the dimension in which points in a region to be split have maximum spread, but chooses the *splitval* to be the mid-point of points' spread in the chosen *splitdim* instead of the median value. The method does not produce totally balanced tree, and does not guarantee O(log$n$) query time. However, as it was noted by Moore in (Moore, 1991), the Median of Widest Dimension

can produce long thin regions for badly skewed data. This happens if, in a region about to be split the points are spread more along one dimension than any other, in which case the Median of Widest Dimension method splits the region only along this long dimension into a series of long thin regions (see Figure 3.3(a)). Moore argued in (Moore, 1991) that his method favours larger squarer regions more than thin ones, which fill up empty spaces in the point space, at the cost of slight imbalance of the tree (Figure 3.3(b) shows an example). This allows fewer leaf regions to be inspected if a query ball overlaps into the empty spaces partitioned by these squarer (instead of the thinner) regions. However, no empirical or theoretical evidence is known to have been given to show that inspecting fewer but larger regions in practice also reduces the query time compared to Median of Widest Dimension. Like the Median of Widest Dimension, this method does guarantee O(n) nodes, but because of the non-logarithmic depth not the O($n$log$n$) construction time.

- **Midpoint of Widest Side**: This method was proposed by Arya et al. (Arya et al., 1998) in conjunction with BBD-Trees, and is included in the ANN Library as one of the construction methods of KDTrees. It select *splitdim* to be the dimension along which the side of the region to be split is widest, and *splitval* to be the midpoint of that side of the region. This, however, sometimes results in splits in which one of the resulting sub-regions is empty. The method, hence, does not produce balanced trees and doest not guarantee either O(log$n$) depth, O(n) number of nodes, or O($n$log$n$) construction time. The method does however, produces larger squarer regions, like the Midpoint of Widest Dimension above, and guarantees that aspect ratio of the regions (the ratio between the longest and the shortest side) is at most 2. This bounded aspect ratio property of MidPoint of Widest Side was shown by Arya et al. (Arya et al., 1998) to be important for achieving O(log$n$) query time in worst case in ϵ-NN searching[1]. This method is not the suggested one for KDTrees by the group of Mount and Arya, and hence was not evaluated for this thesis.

- **Sliding Midpoint (SlMidPt) of Widest Side**: This method was introduced by Mount and Arya in their ANN library to over come the short comings of both Median of Widest Dimension, and Midpoint of Widest Side. It is suggested as the method of choice by Mount and Arya in their ANN library (Mount, 2006). It selects the *splitdim* in the same way as Midpoint of Widest Side, which is the dimension in which the region has its widest side. However, the selection of *splitval* is modified,

[1]The dependence of query time on d is still exponential

if setting it to the midpoint of the region's widest side is resulting in an empty subregion, then the *splitval* is slide towards the non-empty subregion until there is at least one data point on the empty side. The tree produced as such is not balanced, and is not guaranteed of O(log$n$) depth. Also, unlike in the Midpoint of Widest Side, the aspect ratio is not bounded for all regions. The tree, however, has the property that it adopts well to the structure of the data, with most of its leaf regions concentrated towards the data points and thus avoiding the empty spaces (see Figure 3.3(c)), and it has been argued by Maneewongvatana and Mount (Maneewongvatana & Mount, 2002, 2001) that in practice the depth of the tree is O(log$n$). Maneewongvatana and Mount in (Maneewongvatana & Mount, 2002) have also presented an empirical analysis of the method and have shown it to give better query time than the Median of Widest Dimension method if query set Q does not follow the distribution of the data S, but not consistently better if both Q and S are from the same distribution. In (Maneewongvatana & Mount, 1999) Maneewongvatana and Mount have also suggested that the method still has the benefits of Midpoint of Widest Sides' bounded aspect ratio property, even though not all regions in the tree produced have bounded aspect ratio. The method gives O($n$log$n$) construction time in practice, however, it is not guaranteed because of the non-logarithmic depth (Maneewongvatana & Mount, 2002, 2001).

- **Fair and Sliding Fair Methods**: These methods are also included in the ANN library (in the ANN manual (Mount, 2006) referred to as *fair-split* and *sliding fair-split* rules). The Fair method tries to achieve a balanced tree structure while still maintaining the aspect ratio bound on the regions. However, unlike in the Midpoint of Widest Side method, in this method the bound is user specified. The method first selects the dimensions which can be used for splitting without violating the bound, and then selects as *splitdim* the one in which the region has the maximum point spread. The *splitval* is then chosen such that the points are divided as evenly as possible among the two sub-regions, while still maintaining the aspect ratio bound. The method, however, like Midpoint of Widest Side method, can generate empty regions. Sliding Fair Method overcomes the short coming of the Fair Method, using the same strategy as the Sliding Midpoint of Widest Side.

  Note that these two methods are only mentioned for completeness sake here, as they are not suggested to be the best by the group of Mount and Arya, and, apart from the ANN manual (Mount, 2006), no known publication is known to mention their

use. Hence, they were not evaluated in this thesis.

- **KMeans Inspired Method**: This method was developed independently during the research for this thesis. It is based on the well known K-Means clustering algorithm. In a K-Means clustering algorithm the goal is to choose a set of K cluster centres, such that the total sum of squared distances of the data points to their nearest cluster centre is minimum. Mathematically, the goal is to minimize following function J, where c_k is one of the K cluster centres, and x_i^k is the datapoint i that has been assigned to cluster centre c_k:

$$J = \sum_{k=1}^{K} \sum_{i=1}^{n_k} |x_i^k - c_k|^2.$$

In the KMeans Inspired Method, we try to select an axis orthogonal split such that the sum of squared distances of the points to their respective centres on each of side of the split is minimum. To achieve this, the points are sorted along each dimension, and a split value s is selected that minimizes the squared distance of the points to their cluster centres on each side of the splitting plane. The *splitdim* is chosen to be the dimension for which the squared distance of points is minimum, and *splitval* to be the value that was calculated earlier for the selected *splitdim* (which gave the minimum squared distance for that dimension). Mathematically, for each dimension j, after sorting the points along the dimension j, we calculate

$$splitval = min_s(\sum_{i=1}^{s}(x_i - \bar{x}_1)^2 + \sum_{i=s+1}^{n}(x_i - \bar{x}_2|^2),$$

where,

$$\bar{x}_1 = \frac{\sum_{i=1}^{s} x_i}{s},$$

and

$$\bar{x}_2 = \frac{\sum_{i=s+1}^{n} x_i}{s-n},$$

and then select the *splitdim* to be the one for which *splitval* is minimized.

Note, that the sum $\sum_{i=1}^{s}(x_i - \bar{x}_1)^2$ above is

$$\sum_{i=1}^{s}(x_i - \bar{x}_1)^2 = \sum_{i=1}^{s} x_i^2 - s\bar{x}_1^2,$$

and hence, can be calculated with one linear scan of the data points for all split locations s (which is also true for the symmetrical case of $\sum_{i=s+1}^{n}(x_i - \bar{x}_2)^2$). However,

since both x_i and $\bar{x}_1$ are d dimensional vectors, each squared distance above involves another sum over all the dimensions j, and, if x_{ij} and $\bar{x}_{1j}$ are co-ordinates of the j^{th} dimension of x_i and $\bar{x}_1$ respectively, the above actually is:

$$\sum_{i=1}^{s}\sum_{j=1}^{d}(x_{ij}-\bar{x}_{1j})^2 = \sum_{i=1}^{s}\sum_{j=1}^{d}x_{ij}^2 - s\sum_{j=1}^{d}\bar{x}_{1j}^2. \tag{3.1}$$

Hence, as it can be noticed from the above that the construction time of this method is linear in n but quadratic in d, and hence the method is not feasible for higher dimensions. Figure 3.3 (d) graphically illustrates the result of applying this method to an example data.

All of the above methods can be affected by the points' range along each dimension. If one dimension is hugely wider than others, than it would always be used for splitting instead of any other. Hence, to make the trees adapt better to the structure of the data, the points' values along each dimension were normalized (to lie in [0,1]).

Apart from the choice of *splitdim* and *splitval*, the only other remaining factor that affects the construction of KDTrees, is the user specified parameter for bucket size b. It was theoretically analysed by Friedman et al. and was shown in terms of query time to be optimal at $b = 1$. However, since then it has been looked at least by Sproull (Sproull, 1991), and Talbert and Fisher (Talbert & Fisher, 2000), to name few, who have empirically shown it to be not the case. Both Sproull, and Talbert and Fisher, have not suggested any single optimal value, and have only performed their study on a single dataset, searching for a single nearest neighbour ($k = 1$). No other study is known to give either theoretical or empirical suggestions of an optimal value. For this thesis, a single value of $b = 40$ was selected. It was chosen to adequately cover the maximum evaluated k ($= 10$), in order to maximize the probability of finding all the kNNs in the first encountered leaf, thus to maximize pruning and speed up the kNN search. This value is near the optimal value of both Sproull, and Talbert and Fisher, and, as will be seen in the next chapter, produces results which compare well with other NN search techniques.

3.1.4 Query in detail

During backtracking, when searching for k nearest neighbours of a given query, for every internal node encountered during the recursion we only look at the side of its splitting plane that does not contain the query only if the query ball intersects with the splitting plane and overlaps with the region on the other side.

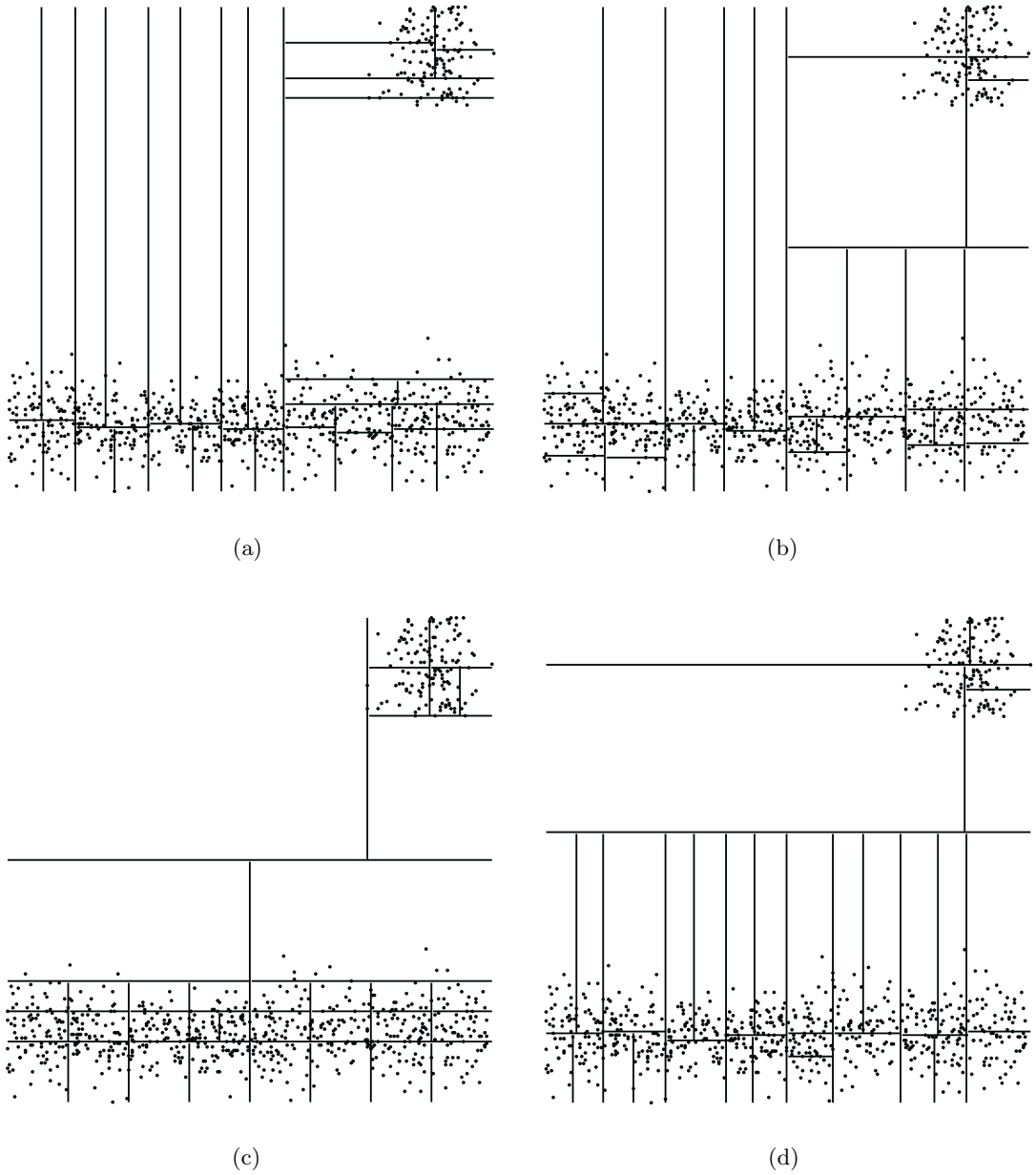


Figure 3.3: KDTree construction methods: (a) Median of Widest Dimension, (b) Midpoint of Widest Dimension, (c) Sliding Midpoint of Widest Side, and (d) KMeansInspired method.

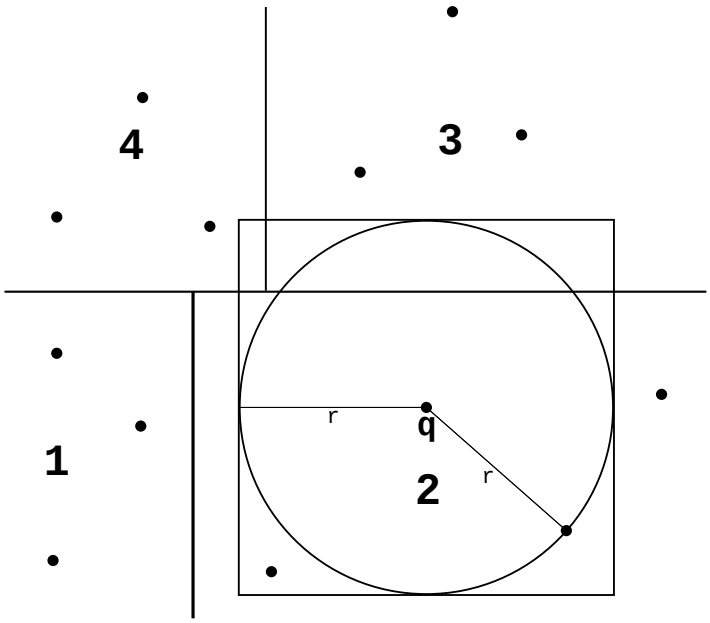


Figure 3.4: Sproull's method for overlap detection. Region 4 is also searched even though it does not overlap with the query ball.

To determine the overlap, different approaches exist. Friedman et. al, in their original version of KDTree, stored the bounds of each rectangular region represented by a node and calculated the exact distance of the query to that rectangle. Sproull suggested an alternative technique to do away with the need of storing rectangular bounds of a region in a node. He suggested to check the overlap only against the hypercube bounding the query ball. This amounts to only looking at the distance of the query point to the side of the cube orthogonal to *splitdim*, and seeing if that distance is greater than the distance to the splitting plane. This involves only looking at the *splitdim* co-ordinate of the query. So, if q_j is the j^{th} co-ordinate of the query point and r the radius of the query ball, then there is an overlap if:

$$|q_{splitdim} - r| \geq |q_{splitdim} - splitval|$$

The method, since it does not calculate the exact distance, has the drawback that those regions that do not overlap with the query ball and hence cannot contain the (k^{th}) NN are also searched (see Figure 3.4).

Arya and Mount in (Arya & Mount, 1993) noted that in higher dimensions, because of the large difference between the volume of a ball and its enclosing hypercube, the Sproull's method can result in visiting a significantly high number unwanted regions. Hence, Arya and Mount suggested another technique that does away with the need of both storing the rectangular bounds of a region as well as doing inexact distance calculation. They introduced in (Arya & Mount, 1993), what they call *incremental distance calculation*. This

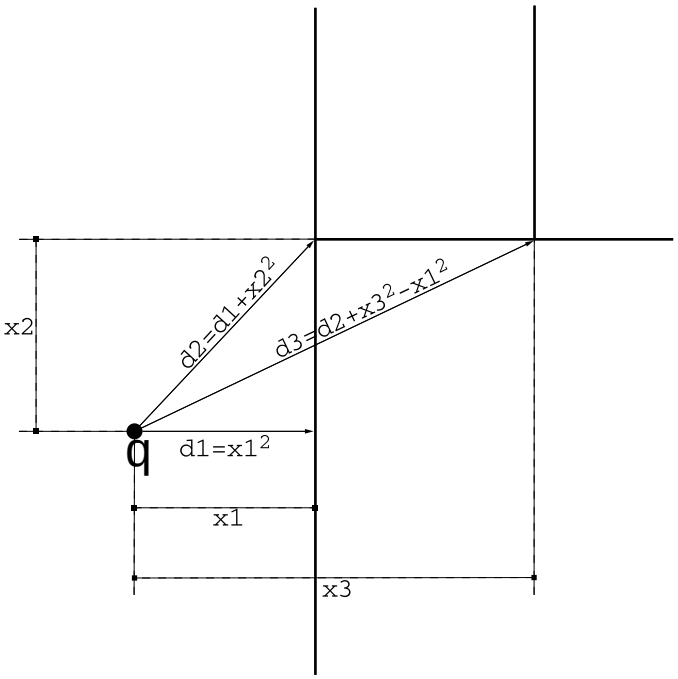


Figure 3.5: Incremental Distance Calculation.

method maintains a variable *dist* throughout the recursive search, and incrementally adds to this variable the exact distance of a region, which intersects with the query ball and needs to be searched. It only stores bounds of a region along *splitdim*, which are required to incrementally calculate the exact distance. The method is illustrated graphically in Figure 3.5. This method is also utilized in the KDTrees implemented for this thesis employed.

3.1.5 Implementation details

For the research of this thesis, the KDTrees were implemented using an array of indices for the data points. Each node of the KDTree stores a start and end index of this indices array, and the subarray defined by those start and end indices store the points contained in the rectangular region defined by that node. Each time a region of a node is split the subarray, defined by the start and end indices stored in the node, is re-arranged such that the points belonging to the left child (those with $splitdim \leq splitval$) are on the left side of the sub-array and the points belonging to the right child (those with $splitdim > splitval$) are on the right side of the sub-array. The start and end indices of the left and right child are then set accordingly after the split. The root node stores the start and end indices that comprise of the whole array, and the whole build process of hierarchically decomposing the point space also hierarchically rearranges the array so that each sub-array (defined by start and end indices in the nodes) contains the points of

each region in the decomposition. The internal nodes of KDTrees, traditionally, only store information on the splitting hyperplane and the bounds of the rectangular region defined by the nodes. They usually do not store any information on the points inside their regions, as it is usually stored in the leaf nodes (e.g. as in the ANN implementation (Mount & Arya, 1997)). However, due to the efficient use of only one single array for the whole tree, the internal nodes of the KDTrees implemented for this thesis also contain references to each of the points inside their region.

During kNN search, the implementation utilizes a priority queue (PQ) to store the k NNs and the distances of the k NNs to the query. The radius of the query ball is taken to be the distance of the top most (farthest) element on the PQ. The PQ is updated each time a better k^{th} NN is found during the search. At the start, the PQ is initialized with k null elements and distances set to positive infinity. This ensures that in case $k > b$ (bucket size) the query ball intersects with all the regions and that we do find k NNs for a given query when $k > b$.

To further speed up the kNN search, the trees were augmented with Partial Distance Search (PDS). The trees use PDS instead of simple linear search when NNs are searched inside leaf regions.

Care was taken to optimize the implementation in every way possible. The implemented trees calculate only the squared distance while searching for kNNs of a given query, and avoid costly square root operations. The actual and squared distances of a given query q increase monotonically, and the costly square root operation is not necessary to determine if one distance is greater than the other as long as both the distances are measured from the same point q. Once the search is complete, the trees take the square root of the distances of only the k NNs remaining at the end, and hence avoid the operation for all the intermediate points looked at (and all the nodes pruned away) during the search. Furthermore, for Median of Widest Dimension construction method, a fast median finding algorithm described in (Manber, 1989) was used that works in O(1) expected time. Its runtime was confirmed empirically on a series of randomly generated numbers, when the algorithm was compared against a few other alternatives in a small scale experiment during the implementation phase of this thesis.

To deal with pathological cases when too many points are co-linear or are too near to each other, the build procedure stops the splitting of a region not only if there are $\leq b$ points in the region, but also if the spread of the points in a region becomes lower than a given threshold. Hence, the implementation stops splitting either if the number of points in a region is falls below $b = 40$ or if the spread of the points (that are inside that region)

along *splitdim* is less than 0.01 (i.e. less than 1% of the spread of all the data points along *splitdim*).

3.2 Metric Trees

Metric Trees like the KDTrees are also binary trees. They hierarchically decompose the point space into hyperspherical regions. Each node of the tree is associated with a single ball representing a hyperspherical region of the point space, and stores that ball's centre and radius. Each node also stores the points that are inside it's ball. An internal node, in addition to these, also stores it's two child nodes which decompose it's ball into two (usually) smaller balls. The balls are allowed to overlap. However, the points can only belong to one ball and only one of it's sub-balls. The trees require only the pair wise distances of the points to be known, that is they only require the distance measure to be a metric, and hence are called Metric Trees.

3.2.1 Basic Construction

Metric Trees, unlike other NN methods evaluated in this thesis, do not have a single basic construction method. In KDTrees, for example, construction involves selecting a *splitdim* and *splitval* for a region and then splitting the region according to these selected values. All the variants of KDTree's construction methods then just deal with the selection of *splitdim* and *splitval*. In Metric Trees, however, unlike the KDTrees and others, there is no such single fundamental method of construction.

Metric Trees have three fundamental construction methods. These are Top Down, Bottom Up and Middle Out construction methods. The description of these methods follows below.

1. **Top Down Construction:** The Top Down construction method, as the name implies, builds the tree from the top node to the bottom ones. This method is somewhat similar to the KDTrees' construction method . We start by assigning the root node a ball bounding the whole point space. The root ball is then split into two (usually) smaller balls which are then assigned to the two children of the root node. The splitting process is then recursively applied to each of the two child balls, and like in KDTrees, the splitting stops when the number of points in a ball falls below a given threshold (similar to the bucket size b parameter in KDTrees). The procedure is illustrated graphically in Figure 3.6. In the ball decomposition in Figure 3.6, the balls of leaf nodes are represented with lighter balls, whereas the darker ones are for

the internal nodes. The numbers inside the balls represent the ball centre and also denote the node/leaf to which a ball belongs in the corresponding tree illustration on the left hand side in the figure.

2. **Bottom Up Construction:** The Bottom Up construction methods builds the tree by building the bottom most nodes first and then iteratively going up building the higher nodes. The method starts by finding from all the points a pair which has the smallest bounding ball, and then creating a node for that pair of points. The method then iteratively finds and builds nodes for a pair of points, or nodes, or a pair consisting of a node and a point, whichever has the minimum bounding ball than the rest of the pairs. This iterative procedure of finding and building of nodes for pairs with minimum bounding ball continues until all points have been merged into nodes, and all nodes have been merged into one top node.

3. **Middle Out Construction:** This method works by finding clusters among the points and then building nodes from those clusters using a method called *Anchors Hierarchy* (discussed below in the next section 3.2.3). These nodes of point clusters are then merged using the Bottom Up construction into one top node. The process of finding clusters, building nodes, and then merging them into one top node is then applied recursively to each of the nodes of clusters created earlier, and the recursion stops if for some node the number of points falls below a given threshold. Figure 3.7 gives a sketch of the algorithm graphically.

3.2.2 Basic Query

In Metric Trees, the hyperspheres (balls) decomposing the point space, unlike the hyper-rectangular regions of KDTrees, are completely closed. The balls corresponding to the leaf nodes are often concentrated towards clusters of points in the point space, and hence it is possible that a given query q does not lie in any leaf region. Hence, for a given query q, the kNN search is carried out by first inspecting the leaf node whose ball is nearest to the query (or whose ball centre is nearest to the query, if both leaf balls contain the query). At each internal node, the search procedure measures the distance of q to the ball centres of each child node, and recursively inspects the child node whose ball (or, in case of both balls containing q, whose ball's centre) is nearer to q. Upon reaching the desired leaf node, the kNNs of q are searched among its points using simple linear search, and are stored. Like the KDTrees, doing so is equivalent to computing the query ball, which is centred at q and has radius equal to the best encountered k^{th} NN. During back

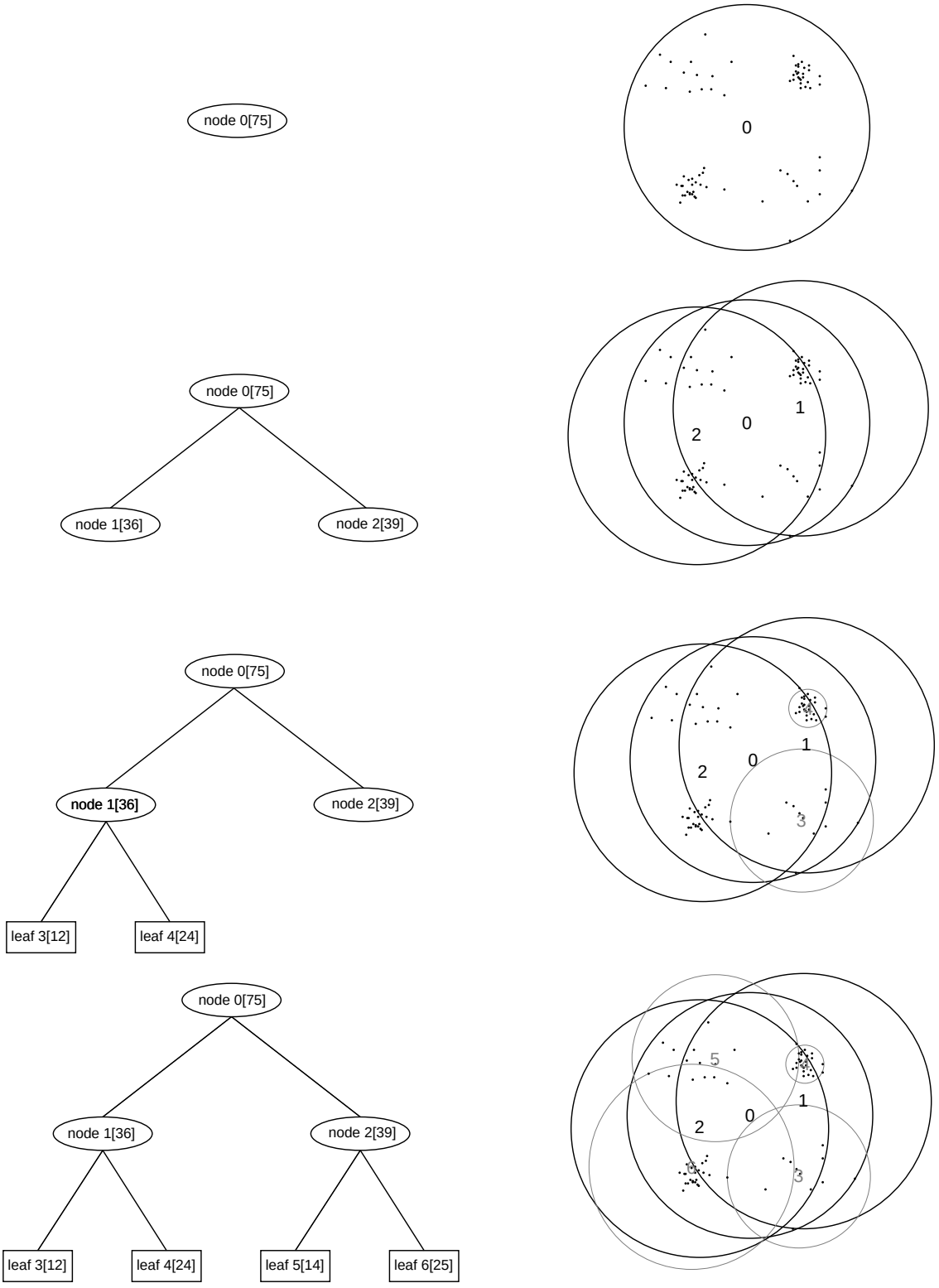
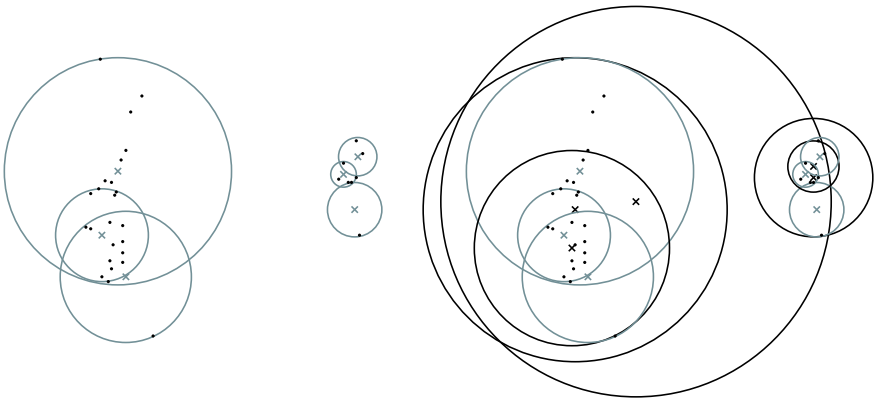
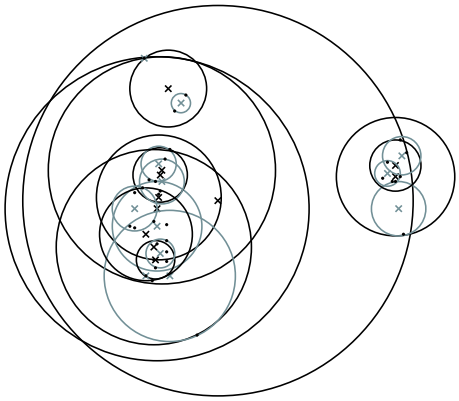


Figure 3.6: Top Down construction method for Metric Trees.



(a) (b)



(c)

Figure 3.7: Middle Out construction method for Metric Trees. (a) Tree after creating cluster nodes, (b) after merging the clusters nodes into one top node, and (c) after recursively applying the process inside each of the cluster nodes.

tracking, at each encountered internal node, the procedure only inspects its other child (whose ball/ball's centre was farther from the query) if the query ball intersects with it. If there is an intersection, then the whole procedure is applied recursively to the other child. Figure 3.8 gives a graphical example of the procedure.

3.2.3 Construction in Detail

Let us now look at the construction if the trees in more detail. Metric Trees since their initial inception, have a number of construction methods proposed for them. Most of the methods are from Omohundro, who is one of the authors who initially proposed the structures, whereas more recently a few have been suggested by Moore. Omohundro, when he presented Metric Trees (he called them Ball Trees) in (Omohundro, 1989), gave five different construction methods for the structures. He also gave an experimental comparison of the his presented methods. He, however, only compared the methods in terms of their construction time and the total volume of balls created, for a given number of data points. He did not evaluate his methods in terms of their query time. While recently, the methods presented by Moore in (Moore, 2000), have not been evaluated against the methods of Omohundro or any other, either by Moore or his group. Uhlmann, the other presenter of the structures, also gave a construction method when he presented the trees in (Uhlmann, 1991a,b). His method has not much received attention of the others, and no study is known to be published by him or anyone else which gives the evaluation of his method against other proposed methods.

The various construction methods for the trees are described in detail below, grouped by their presenters, Omohundro, Uhlmann, and Moore.

Omohundro's Methods

1. **Median of Widest Dimension:** This method is similar to the KDTree's standard construction method (called by Omohundro the *k-d* Construction method), and builds the tree top down. It splits a region according to the median value of it's points in the dimension in which they are maximally spread. After finding the dimension *splitdim* in which the points of a node are most spread and the median value *splitval* along *splitdim*, similarly to KDTrees, this method makes the left ball from the points whose *splitdim* is less than or equal *splitval* and makes the right ball from the points whose *splitdim* is greater than the *splitval*. The Metric Tree constructed with this method is always perfectly balanced, has O(log$n$) depth, and can be constructed in O($n$log$n$) time.

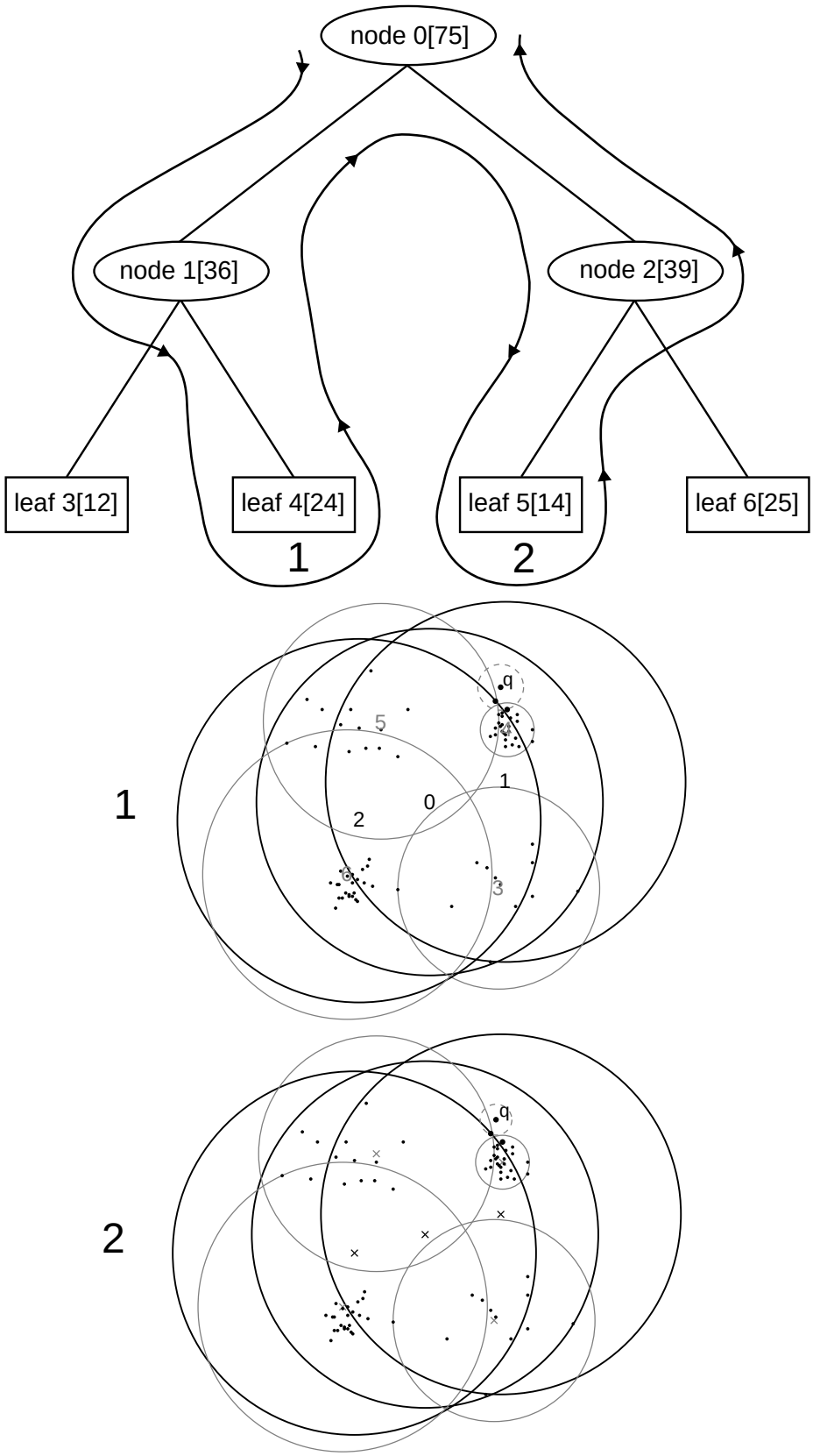


Figure 3.8: Illustration of Metric Trees query procedure.

2. **Online Insertion Method:** This method constructs the tree top down and online by point insertion. It inserts a point into the node which would result in minimum increase in the volume of the tree. For a given point p, it goes through the nodes of the tree, calculating their volume increase and the increase in volume of their ancestors, that would occur as a result of inserting p in them. The method maintains the candidate nodes for insertion of p in a priority queue and inserts p in the best node it finds that gives the minimum volume increase for the whole tree.

3. **Cheaper Online Insertion Method:** This method is similar to the Online Insertion Method. The only difference is that instead of storing the points in PQ and looking at both the branches of the tree at each internal node when searching for the best node for insertion, it only looks at the branch of the child which gives smaller volume increase and does not maintain a PQ of candidate nodes. The best node is found in this heuristic manner and the point is inserted in the best node. The method is cheaper in computational and storage cost as less nodes are inspected and no node needs to be stored during the procedure.

4. **Minimum Volume Increase Heuristic:** This method similarly to Online Insertion Method tries to produce trees with the minimum sum of ball volumes. The method similarly to Online Insertion works top down, however, it is offline and requires all points to be known in advance. For a given node region, this method sorts the region's points along each dimension, and then for each dimension after sorting it goes through the sorted list of points twice, first from left to right and then from right to left. When going from left to right through a list, the method calculates the volume of the ball bounding the points, as each point is scanned (or in other words each new point is inserted into the ball and its volume is calculated). The same is done when doing the reverse pass from right to left. These two passes for a dimension then give the volume of the bounding balls on both sides of each split location of the points along that dimension. The method then uses the two balls which had the minimum sum of volumes, for some dimension and its associated split location, to split the given node. This method is somewhat similar to the KDTree's KMeans Inspired Method. There we find a *splitdim* and *splitval* which give the minimum sum of squared distance for points on both sides, and then split the region into hyperrectangles according to those *splitdim* and *splitval*. Here, we find a *splitdim* and *splitval* which would give the minimum volume of bounding balls for points on both sides, and then split the region into two hyperspheres with the balls

of those *splitdim* and *splitval*. According to Omohundro, the method as such takes $O(n(\log n)^2)$ construction time.

5. **Bottom Up method:** Omohundro's described Bottom Up method works similarly to what has been described earlier in section 3.2.1. It maintains a list consisting of points and nodes, and iteratively finds from the list, a pair of points, or nodes, or a pair consisting of a point and a node, whichever has the smallest bounding ball, and creates a node for that pair with that smallest bounding ball. The method then replaces the pair for which the node is created, with the created node in the list, and iterates again to create a node with the smallest bounding ball. The ball size is measured using the volume by Omohundro. However, since the volume and a ball's radius both increase monotonically with each other, radius is sufficient to measure a ball's size as has been done by Moore in (Moore, 2000). It can be noticed that the process can require upto $\frac{n(n-1)}{2}$ distance computations (for calculating radius of bounding ball) for a single pass for building a node from a pair. Hence, in worst case if $O(n)$ nodes are created, the method can take upto $O(n^3)$ construction time.

Omohundro, when he presented the above construction methods in (Omohundro, 1989), suggested that the Median of Widest Dimension produces the best quality trees (in terms of fitting to the structure of the data, measured by Omohundro with total volume of balls in the tree) when the data is uniformly distributed. However, when the data is clustered or non-uniform, then according to Omohundro, Bottom Up produces the best quality trees followed closely by Minimum Volume Increase Heuristic method. In this thesis, we more interested in the query time performance of a NN search method. However, intuitively if a structure adopts well to the local structure of the data, it is more likely to narrow down to the right region of a query point, and more likely prune away regions unlikely to contain a NN. Hence, based on this intuitive idea, Omohundro's Median of Widest Dimension, and Bottom Up and Minimum Volume Increase Heuristic methods were implemented, as they represented the two ends of the spectrum (of adapting to the structure of the data) and were likely to contain the best candidate in terms of query time performance and would have reinforced this intuitive idea if it were true in practice. The Bottom Up method, which has a prohibitively high construction cost, was not implemented for evaluation but just to compare how well other methods fit the data in comparison with the Bottom Up, as it was the best one that fit to the local structure.

Omohundro in (Omohundro, 1989) used balls instead of points to construct the trees. So, instead of a set of points, he considered a set of balls and built the tree for that set of

balls. Immediately after the Minimum Volume Increase Heuristic method of Omohundro was implemented during the research phase of this thesis, it was noticed that it was producing extremely skewed trees. Each node of the tree was being split with one ball consisting of only one point and the other consisting of the rest of the points. While trying to carefully debug the implementation it was revealed that it was always the case that minimum sum of ball volumes in (almost) every case would only occur if one ball consisted of only one point (and hence have zero volume) and the other of the rest of the points. Though every strife was taken to make sure the implementation as error free as possible and as close as possible to the original description, still the implementation could not be made to work as shown in (Omohundro, 1989). Even though Omohundro in (Omohundro, 1989) used balls instead of points, he still used the degenerate cases where balls consisted of single points, which should in essence be equivalent to what has been performed for this thesis, however, still the method could not be made to work as described. Hence, the full evaluation of Minimum Volume Increase Heuristic could not be performed in this thesis, and only Median of Widest Dimension of Omohundro was fully evaluated against the rest. Preliminary evaluation of the method, whilst it produced skewed trees, also showed that it was not worthwhile to proceed with the full evaluation, as the method performed poorly compared to others.

Uhlmann's Method

1. **Median Distance From Arbitrary Point:** This is a Top Down construction method. It splits a ball of a node into two balls based on the median distance of an arbitrarily selected point to all the other points inside the ball. Hence, after arbitrarily selecting a point p' from inside the ball, the method computes it's distance to all the other points inside the ball, and then computes the median value m of those distances. The method then creates a ball for the left child from all points $p_i|d(p_i, p) \leq m$, and makes a ball for the right child from all points $p_i|d(p_i, p) > m$. The tree as such can thus be constructed with O($n$log$n$) distance computations in O(dnlog$n$) time, which is perfectly balanced and has O(log$n$) depth.

Moore's Methods

1. **Points Closest to Furthest Pair:** This is also a Top Down construction method presented by Moore in (Moore, 2000). It splits a ball based on the pair of points inside the ball which are furthest from each other. Finding the furthest pair in a ball with n points is an O(n^2) process, hence, the method uses a linear time heuristic to

find approximate furthest pair. The method works by finding a point $p1$ which is furthest from the centre of the ball of a given node. It then computes the distance of $p1$ to all the other points inside the ball to find $p2$ which is furthest from $p1$. The method then creates a ball for the left child from the points which are closer to $p1$ (than $p2$), and creates a ball for the right child from points which are closer to $p2$.

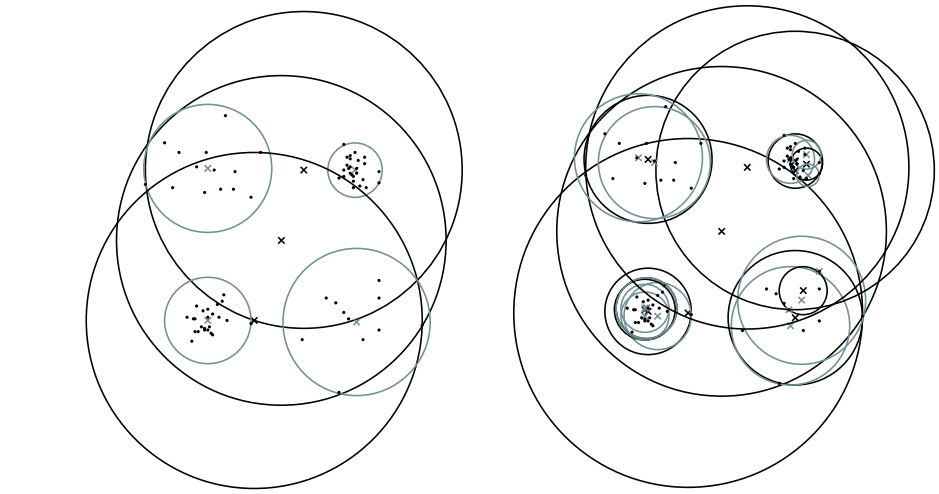
The method has the property that it adopts very well to the structure of the data. Of all the methods only the Bottom Up construction sometimes exceeds this method in this property. The method though works very fast in practice, no theoretical upper bound for it's construction time has been given.

2. **Middle Out method:** Also presented in (Moore, 2000), this method was devised specifically to speed up KMeans clustering algorithm using Metric Trees. The method works by using the Anchors Hierarchy method (also presented in (Moore, 2000)) to first find $\sqrt{n}$ clusters and to make ball nodes out of them. It then uses the Bottom Up method to merge those balls into one top ball, and then recursively applies the Anchors Hierarchy and Bottom Up method to each of the $\sqrt{n}$ balls of clusters initially created.

   The Anchors Hierarchy method works by selecting an arbitrary point, called anchor a_1, from the given set of points and then creating a ball centred at a_1 which includes all the points. The method then takes the point furthest from a_1 as the next anchor a_2, and creates another ball for a_2 assigning it all the points which are closer to a_2 than a_1. The method then iteratively selects the next anchor a_i which is the furthest point in the biggest ball and makes a ball for it assigning it all those points which are nearest to a_i than any other anchor. The iteration carries on until $\sqrt{n}$ balls for a given set of n points are created.

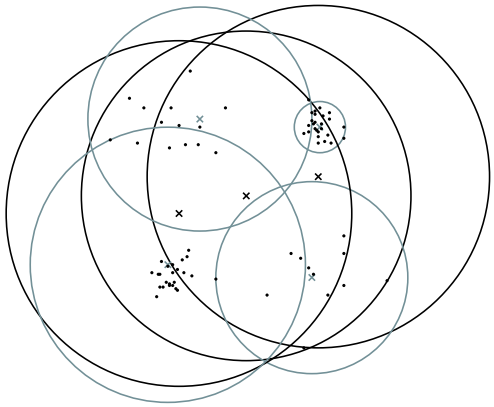
Figure 3.9 graphically illustrates Metric Trees constructed with each of the methods that were evaluated in this thesis; namely, the Median of Widest Dimension, Median Distance From Arbitrary Point, Points Closest to Furthest Pair, and the Middle Out method. Bottom Up method is also included in the figure as it produces the best quality trees in terms of fitting to the structure of the data. It is included to show the quality of the trees produced by the other methods in comparison with the best. It can be seen from the figure that in terms of quality, Bottom Up is followed by, in order: Middle Out, Points Closest to Furthest Pair, Median of Widest Dimension, and lastly the Median Distance From Arbitrary Point method.

All the construction methods mentioned above, from Omohundro, Uhlmann, and

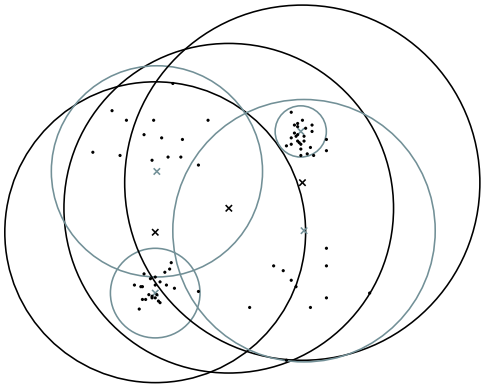


(a)

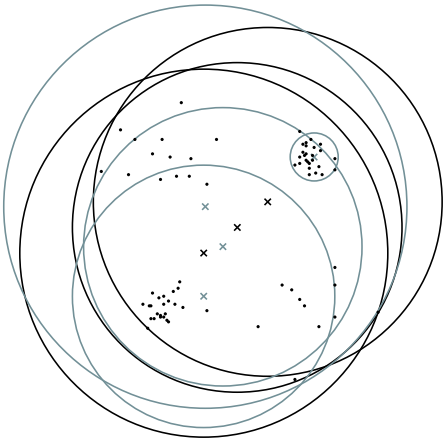
(b)



(c)



(d)



(e)

Figure 3.9: Metric Tree constructed with (a) Bottom Up, (b) Middle Out, (c) Points Closest to Furthest Pair, (d) Median of Widest Dimension, and (e) Median Distance From Arbitrary Point.

Moore, do not have any theoretical worst or expected case upper bound for their query time. Though the trees are said to perform well in practice.

Omohundro, when he presented the Metric Trees in (Omohundro, 1989), kept the discussion general and did not go into the details of the structures. He did not clearly define the notion of a ball's centre. It was also not defined clearly by Uhlmann in (Uhlmann, 1991a,b). For the trees implemented for this thesis, a ball's centre was taken to be the centroid of the points, which is also the centre of points geometrically and has been used by Moore (Moore, 2000). It is calculated in exactly the same way as the centre of points mentioned for KMeans Inspired method for KDTrees in section 3.1.3. Hence, if c is the centre of a ball containing n points, and if c_j is the j^{th} component of the centre and x_{ij} the j^{th} component of a point x_i in the ball, then each c_j is calculated as:

$$c_j = \frac{\sum_{i=1}^{n} x_{ij}}{n} \tag{3.2}$$

The radius of each ball is then set to the distance of the point furthest from the centre c.

Moreover, both Omohundro and Uhlmann in their presentation of Metric Trees, unlike Moore, did not use the notion of maximum leaf size, that is they did not stop their recursive tree construction if the number of points in a ball fell below some given threshold. However, since Omohundro used the concept of balls instead of points, he did imply a maximum leaf size of 1. To make the comparison of the different construction methods, and that of Metric Trees against KDTrees fairer (as KDTrees use bucket size as maximum leaf size), the construction methods of Omohundro and Uhlmann that were evaluated were implemented with the notion of maximum leaf size as the stopping criteria. Hence, for each method used in this thesis the recursive construction stopped if the number of points in a ball fell below (or became equal to) a user specified threshold. The threshold was set to 40 to match the one of KDTrees. Even though, the Bottom Up construction method was not fully evaluated, the notion of maximum leaf size was also applied to it. In the Bottom Up construction, it was applied using a post process procedure that converts all nodes into leaf nodes (by discarding all its descendants), which have less than or equal to the threshold number of points.

Omohundro in his description of the trees, uses balls which bound its child balls. Whereas, both Uhlmann and Moore have used balls which bound the points contained in a ball. Omohundro's technique results in parent balls which are larger and totally contain the child balls. However, in Uhlmann's and Moore's approach the parent balls are smaller as they only bound the points inside both of their child balls. For this thesis, even for the

Omohundro's evaluated construction methods, the approach of Uhlmann and Moore was used, since during kNN search tighter parent balls are likely to result in better pruning. However, in the Middle Out construction method, even Moore has used parent balls which only bound the child balls. This is so because otherwise the Bottom Up process would instead of doing $\sqrt{n}$ distance computations for $\sqrt{n}$ balls of clusters, would end up doing full $O(n)$ distance computations for merging a single pair of balls, and the construction cost, though not as high as $O(n^3)$, would still be $O(n \cdot \sqrt{n} = n^{3/2})$. Hence, in the implementation of Middle Out method in this thesis too, the parent ball during Bottom Up is computed big enough to completely enclose the child balls.

The notion of maximum point spread in leaf nodes, which is similar to that used in KDTrees, also seems to have been used in the implementation of Metric Trees by Moore and his group (available by request from http://www.autonlab.org/autonweb/-10408.html?branch=1&language=2). In addition to maximum leaf size they also seem to be using a threshold (0.01) of ball radius (which essentially reflects the point spread inside a ball) as another stopping criterion for the construction process. As in KDTrees, this is probably also to deal with pathological cases where too many points in a dataset are co-linear. However, this stopping criterion is not mentioned in any of the publications known for the trees by Moore and his group. Hence, the criterion was not used in the Metric Trees implemented for this thesis. Since data exhibiting such pathological cases was also not used for evaluating the NN methods in this thesis, the comparison of the trees with the implemented KDTrees that use this stopping criterion is not unfair. The implementation of the trees, however, is readily modifiable, and this additional criterion can be accommodated with ease if ever required.

3.2.4 Query in Detail

Uhlmann in his description of Metric Trees only considered range queries (points within a specified distance r from the query), and did not provide any procedure for kNN search. Uhlmann and Moore, however, have both considered kNN search and have provided efficient procedures which are essentially the same.

Moore and his group have provided their procedure for kNN search for Metric Trees in (Liu et al., 2004). Their procedure works exactly the same as had been described in section 3.2.2. The criteria for detecting the intersection of the query ball with a node's ball, that their procedure uses to prune away nodes which do not intersect the query ball, is as follows.

Let c_i be the centre of the ball of some node i against which we need to detect the

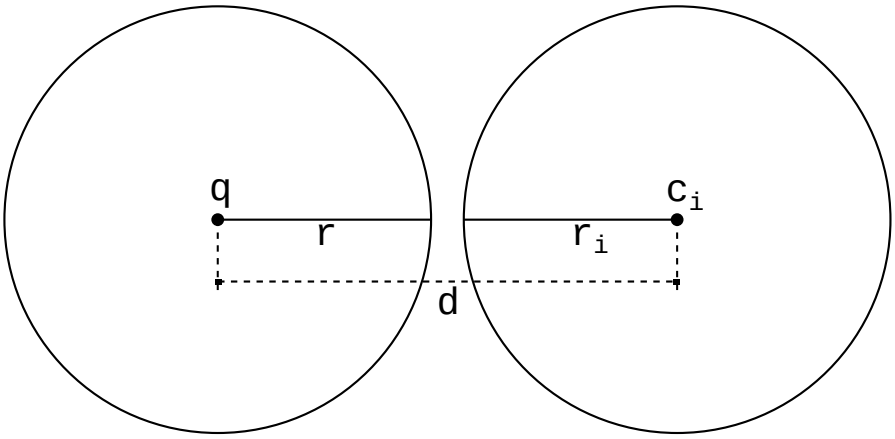


Figure 3.10: Metric Trees' pruning criterion. The node corresponding to ball c_i is only pruned if $r < d - r_i$.

intersection, and let r_i be its radius. Let r be the radius of the query ball, equal to the distance of the best encountered k^{th} NN. The query ball does not intersect with the node's ball, if

$$r < |q - c_i| - r_i, \tag{3.3}$$

and the node can thus be pruned away. However, if $r \geq |q - c_i| - r_i$, then there is an intersection and we would need to search inside the node to be sure we have found the exact kNNs. This pruning criteria is illustrated graphically in Figure 3.10.

3.2.5 Implementation Details

The Metric Trees, like the KDTrees, were implemented using an array of indices for the points. The array is re-arranged in accordance with the build procedure and each node stores the start and end index of this indices array for the portion of the array assigned to its region. The points inside a node are contained in the portion of the array defined by the start and end index stored in the node. Each time a node is split the points belonging to it's left child are moved to left of the node's portion of the array, and the points belonging to the right child are moved to the right in the node's portion of the array. The start and end indices for the left and right child of the node are then set accordingly so that they point to their respective sub-portions in the node's portion of the array.

Like the KDTrees, the k NNs during the kNN search in Metric Trees were stored in a priority queue (PQ). Same as in KDTrees, the PQ was initialized with k null objects and $+\infty$ distances, so that the query ball intersects with all the regions (as the radius of the query ball is taken to be the k^{th} largest distance on the top of the PQ) and we are guaranteed to find k neighbours in case the maximum leaf size parameter is less than k.

After putting in the PQ the first k neighbours found in the first leaf during the search (or the first few if maximum leaf size $> k$), the PQ is then only updated if a better k^{th} NN is encountered in some other leaf region during the search.

The Metric Trees, were also augmented with PDS, which is used in place of simple linear search when k NNs are searched inside a leaf region. The PDS returns $+\infty$ if a point is farther than the current k^{th} NN.

Same as in KDTrees, care was also taken to optimize the implementation of the Metric Trees as much as possible. Only squared distances were calculated during the kNN search to avoid the costly square root operation. The square root of the distances was taken only for the k NNs remaining at the end of the search. However, unlike KDTrees, for pruning away nodes exact unsquared distances are calculated. This is so, because the pruning criteria in Metric Trees involves distances measured from two different points (r and r_c in 3.3), and hence the monotonic relationship between squared and unsquared distances does not exactly hold. Furthermore, for the Median of Widest Dimension construction method, as in KDTrees, O(1) median finding algorithm of (Manber, 1989) was used.

3.3 Annulus Method

The Annulus Method works by projecting the points to scalar values, and then using that scalar projection as the basis of kNN search. The method selects a point as a reference point, and then projects all the data points according to their distances from this reference point. For a given query, the method also projects it to it's distance from the same reference point, and then searches for its neighbours by going in both directions from the position of the query's distance in the scalar projection of the data points. The data points encountered in each direction in the scalar projection are considered as candidate NNs of the query. Projecting the data points as such amounts to approximating their position in the point space as seen from the reference point, and search for NNs of a query using this projection amounts to searching in this approximate space. When the search starts from the position of a query's distance in the projection, it is actually equivalent to looking at points on the circle around the reference point which has radius equal to the query's distance from the reference point. As the search proceeds in both the directions from the position of query's distance in the projection, it actually amounts to expanding this circle in both the directions into an annulus, and thereby examining the points inside that annular region. The search stops when the query ball is completely inside this annular region.

3.3.1 Method's Preprocessing

The method during preprocessing computes the distance of the data point to the reference point and then stores them in an array. The array is then sorted using some efficient sort function. The method thus takes O(n) space and O($n$log$n$) computational time.

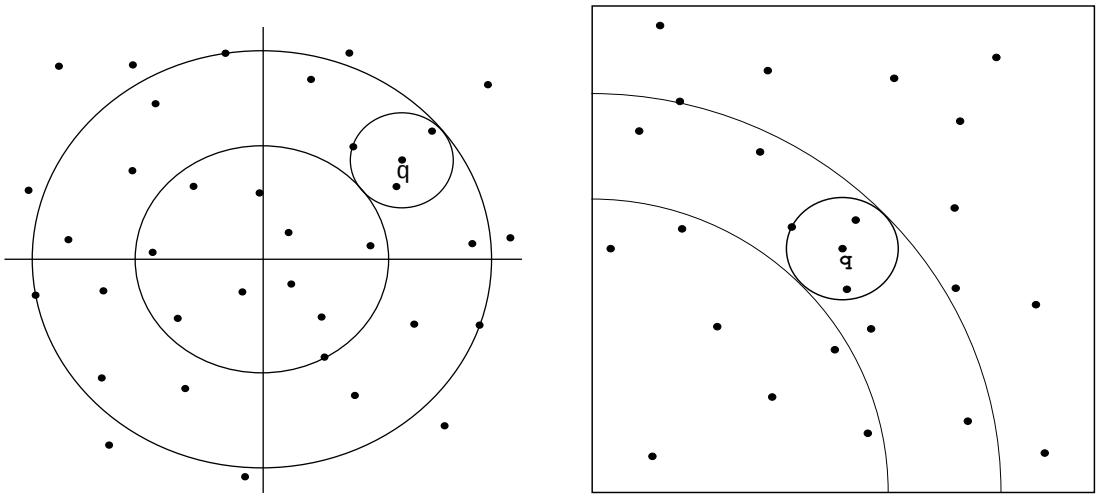
A natural choice for the reference point is the origin or the zero vector. However, during kNN search this can result in looking at points in the whole of the annulus around the origin, especially if the data points are spread in all the four quadrants and the origin is included in the point space (see Figure 3.11(a)). In order to keep the number of points looked at during the kNN search minimum, the Annulus Method implemented for this thesis uses a reference point which is the absolute minimum of the point space. This ensures that the annular region looked at during the kNN search is always confined to the first quadrant, and thereby avoid looking at points which are too far away to be NNs of a given query (see Figure 3.11(b)). If v is the reference point, with v_j the value of its j^{th} dimension, and x_{ij} the value of the j^{th} dimension of point x_i among the n data points, then the absolute minimum v of the data points is defined as:

$$v_j = min(x_{1j},\ x_{2j},\ \ldots\ ,\ x_{nj}).$$

The reference point v can thus be calculated in O(n) time. The total computational cost of the method with the calculation of the reference point, therefore still remains O($n$log$n$) (in actual it is $n$log$n$+n).

The sorting in the Annulus Method implemented for this thesis, is done using an efficient implementation of quicksort. Though, quicksort has an O(n^2) worst case sorting time, in practice it almost always works O($n$log$n$) time. The implementation of the quicksort used in this thesis was empirically found to be faster than the (publicly available) implementations of other sorting methods, in a small scale comparison study during the research phase of this thesis.

The array of distances computed by the implemented Annulus Method, is a 2-dimensional array. It stores the points' distances from v as well as the points' indices, as a reference to the points. The quicksort routine was thus modified to sort this 2-dimensional array according to the distance dimension while still keeping in sync the indices dimension of the array, so that the association of a point with each distance is not lost.



(a) (b)

Figure 3.11: Reference points for the Annulus method. (a) Origin, which can result in a bigger annular region and thus more points being examined. (b) The minimum point in the point space which results in only the annular region in the first quadrant being examined.

3.3.2 Method's Query Procedure

For a given query q, its distance is first calculated from the reference point v. Then binary search procedure is used to find the position of the query's distance in the array of sorted distances of the data points, which was computed during the preprocessing. The search then looks at points, as referred by the indices dimension of the 2-dimensional distance array, in both the directions starting from the position of the query's distance in the distance array. It might be the case, especially if the query does not follow the structure/distribution of the data, that the query point q is either located at (or below) the reference point v, or at (or farther than) the point which is furthest from v. In such a case the position of the query's distance returned by binary search is at the end of the array, and the search for k NNs then just proceeds in one direction. Also, if during the search one side of the array is completely exhausted, in cases where the query's distance is near rather than at the end of the array, then again the search only proceeds in one direction. For each data point looked at during the search, it's distance to the query is calculated, and the point and it's distance are stored in a priority queue (PQ) holding kNNs of the query, if it's distance is less than the distance of the current k^{th} NN on the PQ. Like in the other NN methods of this thesis, the PQ is initialized with k null objects and $+\infty$ distances. The search terminates when the radius of the query ball does not overlap with the points in each direction, that is the query ball is entirely within the annulus

whose width is defined by the distances of the points on each side of our search. The overlap of the query ball is determined as follows. Let x_{lo} be the data point encountered in our search on the low side of the array (side nearer to v), let x_{hi} be the data point on the hi side, and let d_{lo} and d_{hi} be respectively their distances from v. Also, Let d_{qv} be the distance of the query from v. Then the query ball of radius r intersects with the region outside the annulus any one of the following conditions hold:

$$d_{lo} \geq d_{qv} - r \tag{3.4}$$

$$d_{hi} \leq d_{qv} + r \tag{3.5}$$

If none of the above conditions hold, then the query ball does not intersect with the region outside the annulus and is completely with the annulus. At such a point the kNN search can be terminated and the data points found as the k NNs of the query are geometrically guaranteed to be the exact k NNs.

The Annulus Method implemented for this thesis, at each iteration of the search procedure looks at one point in each direction in the array (or only in one direction if only one direction can be searched). It, however, in each iteration keeps looking at the points in one direction as long as the points in that direction are equidistant from v. Points which are equidistant from v are on the surface of the (quarter) circle from v and are also equidistant from the query, and thus all need to be looked at. Doing so ensures that at each iteration we have looked at all the points inside the annular region.

The query time of the method depends on the time to search for the position of the query's distance from v in the distances array, plus the time spent looking at the points during the search using that array. The binary search used for finding the position of the query's distance in the distance array takes O(log$n$) time, while common intuition suggests that the time spent on looking at points in the distance array should also be sublinear in n. Also, intuition suggests, that the time spent on looking at points in the distance array should grow linearly with the dimension d and the method should not suffer from the curse-of-dimensionality.

3.4 CoverTrees

CoverTrees are N-ary trees, where each internal node has an outdegree of $\leq N$. Each node of the tree contains a single point p, and a ball which is centred at p. All nodes

are arranged in levels[2], and nodes at a level i have balls of radius 2^i. The top level of the tree consists of a single node, with a ball big enough to cover the entire set of data points, and its descendants at lower levels have balls of smaller radii that cover the points in finer detail. The lowest level leaf nodes of the tree have balls that, apart from the point contained in the leaf node, do not cover any other point.

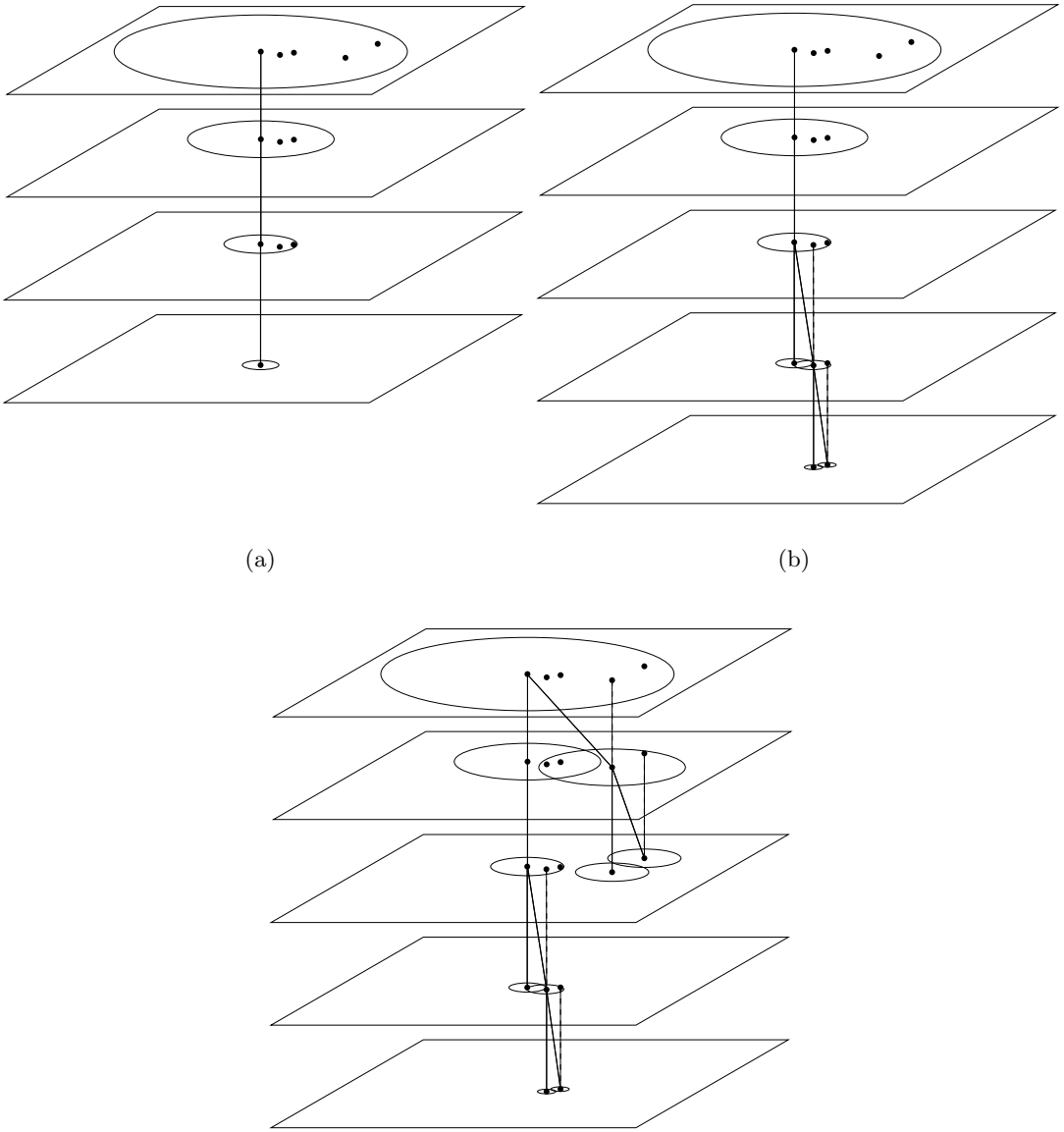
This tree structure, like the Metric Trees, can probably be viewed as a hyperspherical decomposition of the point space. However, it is not exactly a decomposition. The construction process of a Cover Tree follows much tighter set of constraints than the one for structures like Metric Trees. This allows a Cover Tree to have certain properties which are probably not possible in a simple decomposition. These constructions constraints and the properties thus achieved are discussed in the subsections below.

3.4.1 Basic Construction

The build process starts by building the root node with an arbitrary data point p and a ball centred at that arbitrarily selected point p which has radius 2^i, where i is chosen to be big enough to cover the entire set of data points. The same data point p of the root node is then used to build a child node at the next lower level $i-1$ with a ball of radius 2^{i-1}. This process of building child nodes is applied recursively until we reach some level i' at which the the ball centred at p and radius $2^{i-i'}$ does not cover any point other than p. At this stage the created node at level i' is made into a leaf node, and the build process backtracks to the last created node at level $i'+1$. The build process then arbitrarily selects a point p' in the ball of the last created node at level $i'+1$, and recursively applies the whole build process to make finer cover balls for that node's ball starting from point p'. When the build procedure returns from the recursive call for point p', it may have made covering balls for all the points inside the ball of the node at $i'+1$; however, if any points are still remaining then another one is selected arbitrarily and the whole build process is recursively applied again using the new selected point, otherwise the build process simply backtracks to the previous node at level $i'+2$. The build process is applied recursively in a similar manner to all nodes created with p at levels $> i'+1$, as the build process further backtracks. Figure 3.12 graphically illustrates the tree building process. It shows the structure of the tree at the end of first, second, and third and final branch of the recursive construction process (branches from the main initial branch).

The tree constructed as such satisfies the following three constraints:

[2]The notion of levels here is different from the classical notion of levels in tree data structures. Here levels have the same meaning as the scales/levels in Navigating Nets (Krauthgamer & Lee, 2004)



(a) (b)

(c)

Figure 3.12: Illustration of Cover Trees' construction process. Tree at the end of (a) first branch of recursion, (b) second branch of recursion, and (3) third and final branch of recursion.

1. **Nesting:** The set of points stored in the internal nodes at any level i of the tree is a subset of the points stored in the internal nodes at level $i-1$. Hence, if C_i is the set of points stored in the internal nodes at level i, then: $C_i \subset C_{i-1}$

2. **Separation:** For any points p, q stored in the nodes at level i, $d(p,q) > 2^i$. Thus, no point stored in a node can be inside the ball of another node at the same level.

3. **Tree:** Any node (and the point stored in that node) at level $i-1$, has exactly one parent at the previous level i, and hence the graph rendered by the construction is a tree.

Furthermore, it can be noticed that in the tree constructed with the above procedure, the radius of the top most ball depends on the maximum of the inter-point distances, d_{max}. Whereas, the radius of the balls at the bottom most levels depends on the minimum, d_{min}, of the inter-point distances. Hence, the depth of the tree depends on the ratio $\Delta = d_{max}/d_{min}$ between the maximum and minimum inter-point distances, and is at most O(logΔ). If the dataset, on which the tree is built, follows the growth bound of Karger and Ruhl then the depth of the tree is O($c$log$n$) (more detail on this later in section 3.4.3).

The original description of the Cover Trees and Navigating Nets, by (Beygelzimer et al., 2006) and (Krauthgamer & Lee, 2004) respectively, is somewhat more complex than the one that has been presented above. Both (Beygelzimer et al., 2006) and (Krauthgamer & Lee, 2004) have presented a more abstract description of the structures by considering an infinite number of levels (from $-\infty$ to ∞), while in the above and the rest of the description below we only consider the levels explicitly necessary to build the tree. The description here is more concrete and should be easier to view in terms of implementation of the structures.

3.4.2 Basic Query

For 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 child nodes for inspection at the lower level $(i-1)$ whose points are inside the query ball. The radius of the query ball at each level i is set to the distance of the current best k^{th} NN (found from among the points stored in the nodes that have so far been inspected) plus the radius of the balls at level i (2^i). This amounts to shrinking the query ball as we go down the levels, and adding only those child nodes for inspection at the next level whose ball centres (points stored in the nodes) are within the query ball. The search stops when at some level the inspected nodes are all leaf nodes

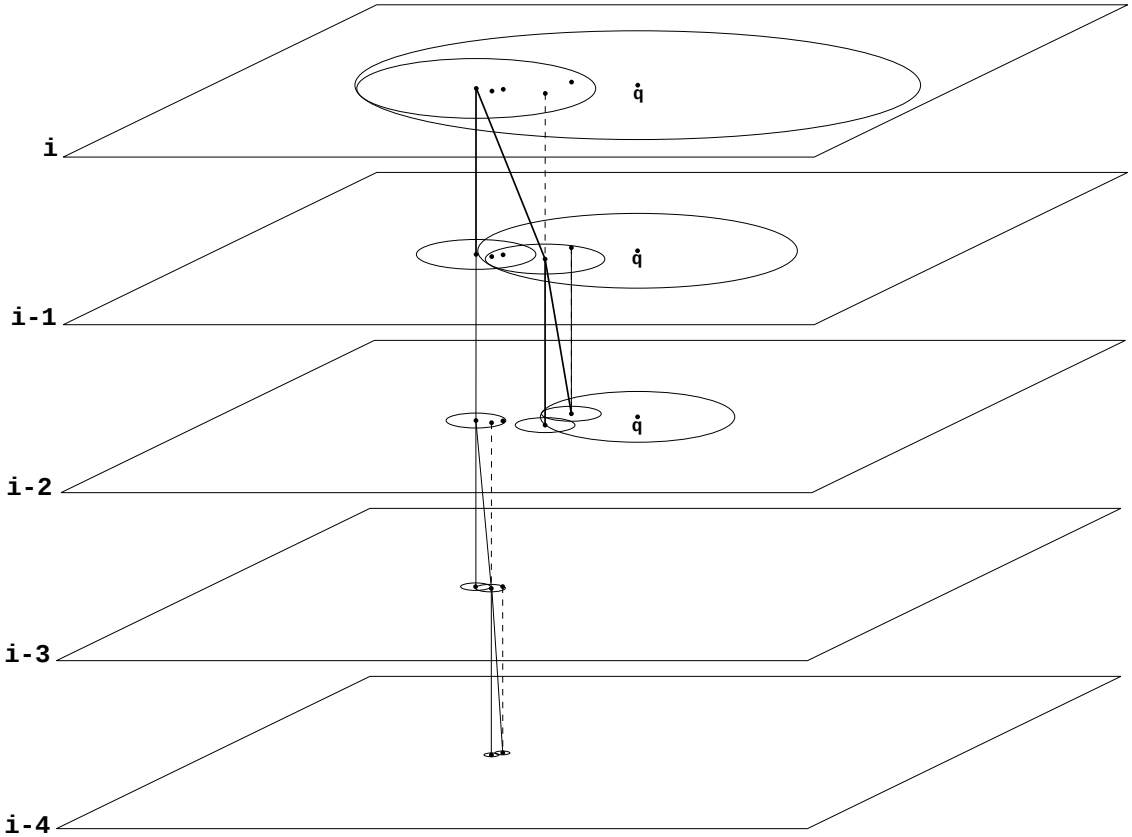


Figure 3.13: Illustration of Cover Tree query.

and have no children. At this stage the best encountered k^{th} NN is the exact k^{th} NN of the query.

Figure 3.13 illustrates graphically the query procedure for 1-NN on the cover tree of Figure 3.12. We start from the root node, going down the branches of all it's children, as they are all inside the query ball. At the next level, $i-1$, we only go down the branches of those nodes whose points are inside the shrunk query ball. Then, at level $i-2$ the search is terminated, as all the nodes being inspected there are leaf nodes, and the NN found from the nodes so far inspected is reported as the NN of the query.

It can be noticed from the above brief description, that the kNN search in a Cover Tree is fairly different than the one in structures that hierarchically decompose/partition the point space (like Metric Trees and KDTrees). First of all there is no backtracking in the search, we only follow one line of search which is guaranteed to return the exact k NNs. Secondly, we do not look into regions which simply intersect with the query ball. We only look inside the ball of a node, if its ball's centre, that is the point p stored in the node, is inside the query ball. These properties are only achievable because of the tighter set of constraints that are imposed upon the Cover Trees during construction. For an insight into how these properties hold for kNN search consider Figure 3.14. It shows

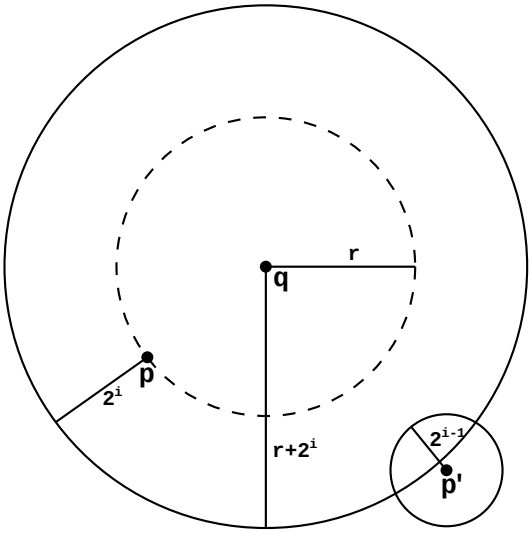


Figure 3.14: Pruning during a Cover Tree's query procedure.

the query ball for a query q at level i. The point p in the figure is the current best k^{th} NN, found in a node at level i, which has distance r to the query. p' is the point of child node under consideration. The radius of the query ball is set to r plus 2^i (where 2^i is the radius of the balls at level i). Note, that the ball of the node for point p' can only contain a neighbour nearer than p, if it intersects with the ball of radius r centred at q. For this to occur the ball of the node for point p' needs to be at least completely inside the the annular region of width 2^i centred at q, thus implying that the point p' be inside the query ball of radius $r + 2^i$. This would always be true regardless of the distance r of the query to the current best k^{th} NN, as the width of the annular region would always be twice the radius of the balls of lower level being considered for inspection. Hence, we can ignore all balls of nodes that even though intersect with the query ball but do not have their ball centres inside the query ball. This would not have been possible if balls at level $i-1$ were of arbitrary radius, as can happen in Metric Trees, but only if the balls at lower levels are of fixed predetermined smaller radius. Moreover, it would probably be easier to see Figure 3.13 in conjunction with Figure 3.14 that as we go down the levels shrinking the query ball from a volume big enough to cover the entire point space, we cannot miss any k^{th} NN of a given query.

3.4.3 Structures in detail

Let us consider the growth bound of Karger and Ruhl again (Karger & Ruhl, 2002). Let $B(p, r)$ denote a ball of radius r centred at p, $|B(p, r)|$ the number of points in that ball, and C_i the set of points stored in the nodes at level i of a cover tree. The growth bound

of Karger and Ruhl is:

$$|B(p, 2r)| \leq c \cdot |B(p, r)|,$$

where c is the *expansion rate* or the *expansion constant* of the dataset $(S \cup Q)$. The growth bound of Krauthgamer and Lee defined in (Krauthgamer & Lee, 2004) (see section), on which the Cover Trees are based, is more general than the one above by Karger and Ruhl. It, however, as noted by Beygelzimer, Kakade and Langford in (Beygelzimer et al., 2005), does not have strong provable results, and the ones that are present are only applicable to ϵ-NN search. Since the growth bound of Karger and Ruhl above is a subclass of the bound of Krauthgamer and Lee, Beygelzimer, Kakade and Langford have presented their results and analyses of the Cover Trees based on the above growth bound.

Beygelzimer, Kakade and Langford in (Beygelzimer et al., 2005), have provided an extensive theoretical analysis of the Cover Trees based on the expansion constant c of Karger and Ruhl's growth bound. If the dataset satisfies the above bound then a Cover Tree, by the procedure mentioned above in section 3.4.1 (the batch construction method in (Beygelzimer et al., 2005)), can be constructed in $O(c^6 n \log n)$ time. The tree only takes $O(n)$ space. The depth is bounded by $O(c^2 \log n)$, whereas the width (defined as the maximum number of children of a node) is $O(c^4)$. The query time required by the tree is $O(c^{12} \log n)$. If no assumption about the growth of a dataset is made, that is the growth bound of the above is not considered at all (or if the c is too large), then the time required to construct the trees is $O(n^2)$. The space requirement of the trees though still remains $O(n)$. The Cover Trees improve on Navigating Nets in this respect, as the Navigating Nets in addition to $O(n^2)$ construction time also require $O(n^2)$ space if no assumption about the growth of the dataset is made.

Beygelzimer, Kakade and Langford in (Beygelzimer et al., 2006) and (Beygelzimer et al., 2005) have also analysed the relevance of the expansion constant c as the basis of analysis of Cover Trees. They have shown empirically for two real world datasets having the same worst case expansion constant c, that the speed up achieved on subsets of the same size of the two datasets is greatly different. This is due to the fact that the distribution of the rate of expansion c of a dataset can be different across points (probably due to the skewed distribution of the dataset). However, since c is not required as a parameter for the construction of the trees, even with different distribution of c among the points the trees can still provide optimum performance, in terms of construction and query time, if used in conjunction with ML/statistical techniques like cross validation and random subsampling, where different subsets of the data are used.

The construction method mentioned above in section 3.4.1 is an offline construction method, which requires a batch of points for construction (hence called batch construction in (Beygelzimer et al., 2005)). An online method based on single point insertion is also given in (Beygelzimer et al., 2006) and (Beygelzimer et al., 2005). It is similar to the kNN query method described above (in section 3.4.1)and works as follows. For a given point p, we go down the levels of the tree while shrinking the ball of p. At each level i, the radius of the ball of p is set to 2^i, and we only add those nodes for consideration at the next lower level, whose points are inside the ball of p. When we reach a level where no points are inside the ball of p, we go back to the previous level select an arbitrary point p' from inside the ball of p and insert p as a child at the lower level in the node of p'. Note, that if p' is inside the ball of p, then p is also inside the ball of p', and hence can be a child of p' while still satisfying the Cover Tree's construction constraints. A single point insertion, as described, takes $O(c^6 \log n)$ time. A single point removal method has also been provided by the authors the trees. It is similar to the described point insertion method and works in $O(c^6 \log n)$ time.

In all of the above we have considered nodes having balls of radius of the form 2^i. This, however, need not be the case, and any base other than 2 can work with the trees (i.e. any radius of the form b^i). According to the authors in (Beygelzimer et al., 2005), a smaller base gives better performance in practice, and the results that have been reported for the trees in (Beygelzimer et al., 2006, 2005) are with a base smaller than 2. The authors, however, have not suggested any optimal value for the base, and have not given any theoretical or empirical analysis of the effect of different bases on the trees. Moreover, the base used for the results reported in (Beygelzimer et al., 2006, 2005) has also not been given. The implementation of the trees used for this thesis uses a base of 1.3, which is the default value in the implementation of the Cover Trees made available by the authors.

For the purposes of clarity a small detail has been left out from the discussion of the cover trees so far. A child of a node at level i does not need to be at the next lower level $i-1$. Infact, after creating the child node at level $i-1$, if the maximum inter-point distance in it's ball is less than 2^{i-1} and a ball of radius 2^{i-2} or smaller can completely cover the points inside the ball, then the radius of the child's ball is set to the minimum $2^{i'}$ so as to cover all the points, and the child node then is made at level i' (where $i' < i-1$). This makes the ball of the child node as small as possible, while still allowing the pruning method mentioned in section 3.4.2 to be applicable. Furthermore, another detail that has been left out is that the leaf node does not need to contain a ball at all. In the original description of the trees the leaf nodes are considered to be at the lowest level

possible ($-\infty$), and hence have the smallest of the balls. For easier comprehension of the structures, this detail was deliberately left out in figures 3.12 and 3.13 where leaf balls at level $i-2$ are of the same radius as the other ball at that level. In actual practice the leaf nodes do not need to have balls and are implicitly considered to be at the lowest level with the smallest balls, even though being much higher in depth in the actual tree structure. This detail was left out, because if single points inserted in the trees (using the procedure mentioned above) end up being inserted beneath a leaf node, then that leaf node is converted into an internal node at the level at which it is located in the tree. Explaining the single point insertion method in a comprehensible manner would have been much harder, if this small detail was not left out.

Moreover, in the implementation of the Cover Trees made available by the authors, the pruning seems to be more stringent than what has been described in section 3.4.2 (or by the authors in (Beygelzimer et al., 2005, 2006)). In the implementation the authors seem to be pruning away the nodes not lying in the query ball, at the level of those nodes. So, instead of checking if the points at the level $i-1$ are inside the query ball of radius $r+2^i$, the implementation checks if they are inside the query ball of radius of $r+2^{i-1}$. The pruning criteria of section 3.4.2 still holds in this case. Still, in this case the ball centre of a node at level $i-1$, needs to be inside the query ball (at least on it's boundary) of radius $r+2^{i-1}$, in order to intersect with the query ball of radius r and thus contain a neighbour nearer than r to the query. In this case the query ball is much tighter and can hence result in better pruning.

3.4.4 Implementation details

The Cover Tree implemented for the research of this thesis is a straight forward Java translation of the C++ code made available by the authors (Available from http://www.hunch.net/~jl/projects/cover_tree/cover_tree.html). The Java translation, however, did not include the very low level optimizations that were used by the authors in their code. The authors in their C++ code have sometimes used CPU registers to store variables, while throughout their program seem to be using data vectors' with lengths padded to CPU's word boundary. At places the authors also seem to be using code specifically for Intel's SSE optimizations. Not all of these optimizations are available in Java. Furthermore, implementing even some of these would have made our comparison of Cover Trees somewhat unfair with other NN methods, since none of these were not included in the implementations of the others, which were implemented much earlier than the Cover Trees during research. Therefore, these low level optimizations were not included.

The implemented Cover Trees are augmented with PDS. This was actually present in the C++ code of the authors, and was included during the translation. During the evaluation phase, however, a bug was discovered in the author's translated code. In the PDS of the authors code, when an accumulated sum for a point becomes larger than the provided sum of the k^{th} NN, it simply returns the square root of the sum as it is, whereas in the PDS that complements implementations of other NN methods in this thesis, a value much larger ($+\infty$) is returned. The problem arises due to numerical imprecision when taking the square root. In the PDS of Cover Tree's authors, in some cases, if the accumulated sum in only slightly larger than the sum of the k^{th} NN, then taking square root makes it equal to the square root of the sum of the k^{th} NN. Hence, a point which is farther than the k^{th} NN and should be eliminated is still included. This problem was corrected when it was found during the evaluation phase, and the results reported in this thesis were generated with the corrected version. The PDS of the Cover Trees was corrected in line with the PDS of the other methods, to return $+\infty$ when a sum became larger than the sum of the k^{th} NN.

In the construction and point insertion methods mentioned in (Beygelzimer et al., 2005, 2006) and the above, an arbitrary point is selected whenever more than one choice of points is available. In the implementation made available by the authors, and hence the one used in this thesis, only the construction method (offline one) is available and it always selects the first point from the set of points whenever more than one choices are present. It thus works in a greedy manner, and builds the structure with the first choice available.

Chapter 4

Experimental Evaluation

In the evaluation phase, the various construction methods of KDTrees and Metric Trees were first compared against each other. These two trees with their best construction methods were then compared against each other, and against Annulus Method and the Cover Trees.

The various construction methods of KDTrees, mentioned in Section 3.1.3, have not been empirically compared with each other in any published study known for the trees. Although people have compared the construction methods of KDTrees before (Talbert & Fisher, 2000; Maneewongvatana & Mount, 2002), even some of the ones mentioned in Section 3.1.3 (in (Maneewongvatana & Mount, 2002)); however, not all the methods of Section 3.1.3 are known to be compared. In case of Metric Trees, only Omohundro (Omohundro, 1989) has presented a comparison of their construction methods; however he only compared the methods that he himself had presented, and no published results are known for any of the other construction methods, mentioned in Section 3.2.3, that were presented later by other people. Table 4.1 gives in summary the construction methods of KDTrees and Metric Trees that were compared against each other in this study.

The popular NN search methods KDTrees and Metric Trees, are also not known to be empirically compared against each other before. Though Marshall is known to be currently working on evaluating the two against each other (along with a number of other methods), his preliminary report (Marshall, 2006) suggests the scope of his comparison is limited only to a particular domain, and also he does not seem to be looking at the various construction methods proposed for the two trees. Cover Trees, since they are one of the most novel of the proposed techniques and were not compared against other popular methods by their authors, still remain to be compared against KDTrees and Metric Trees. The Annulus Method has only known to been compared against KDTrees (Zatloukal et al., 2002) but not Metric Trees.

In the comparison of these NN search methods, simple linear search augmented with PDS was also included, to serve as a baseline. It was included to assess how well complex

KDTrees' construction methods compared
1. Median of Widest Dimension (Median)
2. Midpoint of Widest Dimension (MidPt)
3. Sliding Midpoint of Widest Side (SlMidPt)
4. KMeans Inspired Method (KMeans)

Metric Tree's construction methods compared
1. Median of Widest Dimension (MedianValue)
2. Median Distance From Arbitrary Point (MedianDistance)
3. Points Closest to Furthest Pair (FurthestPair)
4. Middle Out

NN methods compared
1. KDTrees (with SlMidPt)
2. Metric Trees (with MiddleOut)
3. Annulus Method
4. Cover Trees
5. Linear search with PDS

Table 4.1: Evaluated methods.

and specialized NN search methods perform compared to having no specialization at all, and to check whether it is worthwhile to use specialized methods in every situation. The NN search methods compared are also given in summary in Table 4.1.

All of the (specialized) methods that are evaluated are roughly equivalent in preprocessing time and space requirements. Hence, the main comparison of the methods was in terms of their query time performance. However, evaluation of preprocessing time is also included, for the sake of completeness and to demonstrate the effect of constant factors hidden by the O notation. All the evaluated methods, as mentioned earlier in the last chapter, were augmented with PDS to further boost their query time performance.

The next section (4.1) describes in detail the procedure that was used to evaluate the methods against each other. It also describes in detail the experiments that were performed to compare the methods. The section next (4.2) outlines the datasets that were used in the experiments performed, and the results of those experiments are presented in summary in the last section (4.3) of this chapter. Note that only selected results are presented, which best show the overall trends, and are hence representative of the overall set of results. Some additional results are also included in the appendices, to support the ones in the main text.

4.1 Evaluation Procedure

Since we are interested in the query performance of a method. Each method's query performance was assessed for increasing data size and increasing dimensionality, for a fixed query size. The query size was fixed at 1000, and the data size (n) and dimensionality (d) were initially doubled and then increased to a much higher value in the end. The following sizes of n and d were used:

$$n = 1000, 2000, 4000, 8000, 16000, 100000,$$
$$d = 2, 4, 8, 16, 32, 80.$$

For each combination of values of n and d above, datasets were synthetically generated and each method was assessed on a dataset using a separately generated query set of the same d and of fixed size 1000. Assessing each method as such, in effect gave the performance measurement of the method for increasing n for each d, and for increasing d for each n. For each combination of n and d, the datasets were generated synthetically from a number of selected distributions (described in detail in the next section), and for each combination of distribution (D), n and d, each method was evaluated 5 times and the results were averaged over those five runs. The total number of datasets generated, hence, was the product of the number of runs (providing a different random number seed), the number of evaluated distributions (described in the next section), the number of evaluated data sizes, and the number of evaluated dimensions, i.e. $5 \times |D| \times |n| \times |d|$. For each of generated dataset, two query sets were generated (both having the same d as the data and both of size 1000), one with the same distribution as that of the data, and one with uniform distribution. Each method was assessed on both these query sets for each dataset, to see how well the method performs when the queries do and when they do not conform to the distribution of the data. Hence, the overall evaluations carried out gave the average measurements of a method for increasing n for each d, and for increasing d for each n, for each distribution when the query does and does not conform to the distribution of the data.

For each NN method the measurements given in Table 4.2 were made for each generated dataset (which were in total $5 \times |D| \times |n| \times |d|$). In addition to these, for tree based methods, i.e. for KDTrees, Metric Trees and Cover Trees, the measurements given in Table 4.3 were also made. These whole set of measurements on all the generated datasets, for a NN method, constituted a single experiment, and the experiment was repeated for each value

1. Elapsed preprocess time	7. Min points visited
2. Elapsed query time	8. Max points visited
3. CPU preprocess time	9. Total Coords looked at per point
4. CPU query time	10. Avg. Coords looked at per point
5. Total points visited	11. Min Coords looked at per point
6. Avg. points visited	12. Max Coords looked at per point

Table 4.2: Measurements made for each NN method on each generated dataset.

1. MaxDepth	7. Max internal nodes visited
2. TreeSize	8. Total leaves visited
3. NumLeaves	9. Avg. leaves visited
4. Total internal nodes visited	10. Min leaves visited
5. Avg. internal nodes visited	11. Max leaves visited
6. Min internal nodes visited	

Table 4.3: Additional measurements made for tree based NN methods.

of k (number of NNs) that was evaluated. The values of k that were selected for evaluation, are the following:

$$k = 1, 5, 10.$$

In Table 4.2 preprocess times (both elapsed and CPU) are the times taken by the construction/preprocessing procedure of the evaluated NN search method, and the query times (both elapsed and CPU) are the times taken by the NN search method to return the kNNs of the whole query set. The elapsed times are the elapsed clock times for the NN search method, whereas the CPU times (which are available with `ThreadMXBean` class in `java.lang.management` package in the newer Java 1.5) are the times actually spent by the CPU executing the NN search method. The total, avg., min and max points visited are respectively the total, average, minimum and maximum datapoints looked at per query by the NN search method for the given query set. The total, avg., min and max coords looked at per point represent respectively the total, average, minimum, and maximum coordinates/dimensions inspected by the PDS (augmented to the evaluated NN search method) for each visited datapoint for the whole query set.

In Table 4.3, MaxDepth refers to the maximum depth of the tree NN data structure, TreeSize refers to the number of its internal and leaf nodes, and NumLeaves the number of only its leaf nodes. Total, avg., min and max internal nodes visited represent respectively the total, average, minimum and maximum internal nodes visited in the tree during kNN search for the whole given query set, and likewise total, avg., min and max leaves visited are the total, average, minimum and maximum leaves visited during kNN search for the whole of the given query set.

The analysis of the query performance of the NN methods was performed only with the measures CPU query time, and Avg. Points Visited. Whereas the preprocessing was analysed only with the CPU preprocess time measure. It was originally intended to perform a deeper analysis of the methods using the other measures. However, the sheer volume of results produced from the evaluation procedure discussed above made the task a lot less tractable. This can be noticed from the volume of only the representative results placed in the main text in Section 4.3. A brief shallow analysis of the other measures that was performed also suggested the selected measures to be good indicators of the methods' performance. Also, because of the large volume of output and to keep the analysis focused on the main objectives of this thesis, the analysis of the combining effect of PDS with other NN methods, using the Coords looked at per point measures (in conjunction with others), could not be performed.

For all the evaluations that were carried out, each dimension of the generated data and query sets was normalized (to lie in [0,1]). It was done just before calculating a distance (both squared and unsquared) between points, and was done to ensure that in the calculated distance the contributions of the dimensions along which the points are most widely spread do not dominate the contributions from the other narrower dimensions. It can be the case in practical datasets that one or more dimensions are much wider (have a higher range) than the rest of the dimensions, whereas the data is more clustered/structured in those narrow dimensions. Hence, in such cases the contributions to the distance of the narrower dimensions carry more information with regards to the NN problem than the contribution of the dimension with the widest range. However, this is domain dependant, and sometimes the contribution due to the wide range of a dimension is also equally important. For this thesis, since the aforementioned rationale applied fittingly to the generated data and query sets, it made more sense to normalize.

For all the evaluations, KDTrees and Metric Trees had their maximum leaf size parameter set to 40. Whereas, the maximum point spread in a leaf for KDTrees was set to 0.01. A parameter with similar notion of maximum point spread was not necessary for Metric Trees (and was not implemented, as mentioned in the last chapter) as the evaluation did not include pathological datasets with many co-linear points. For Median Distance From Arbitrary Point and Middle Out construction methods of Metric Trees, which require selecting an arbitrary (on random) point from a given set of points, the random seed parameter was arbitrarily set respectively to 17 and 1. The base parameter for Cover Trees was set to 1.3, which is the default value in the version implemented by their authors. Details of these parameters can be found in the relevant sections of the last

1. Uniform	10. Straight Line with Noise
2. Gaussian	11. Two Straight Lines
3. Laplacian	12. Two Straight Lines with Noise
4. Correlated Gaussian	13. Plane
5. Correlated Laplacian	14. Plane with Noise
6. Clustered Gaussian	15. Two Planes
7. Clustered Orthogonal Flats	16. Two Planes with Noise
8. Clustered Ellipsoids	17. Periodic Functions
9. Straight Line	18. Periodic Functions with Noise

Table 4.4: Distributions on which the NN methods were evaluated.

chapter.

All the evaluated NN methods, their supporting classes, and the classes for the evaluation procedure, were implemented within the framework of Weka (Witten & Frank, 2005). The evaluation was carried out using Weka's Experimenter environment, and each experiment was distributed among three identically configured machines. It can be noticed that the values of n and d that are evaluated are rather modest. It ought to be mentioned that initially it was intended to evaluate n upto 10,000,000 points and d upto 1024 dimensions. However, the extra memory overhead associated with the `Instances` and `Instance` classes of Weka, which respectively represented data/query sets and single data/query points, did not allow experiments to be performed with such high values of n and d, even on machines with a sizeable 1GB of RAM. The machines on which the experiments were performed were identically configured with P4 3.0GHz single CPU (with HyperThreading technology)and 976MB of RAM, that ran Gentoo version 1.6.13 of linux and J2SE version 1.5.0_10.

4.2 Evaluation Datasets

The data and query sets on which the methods were evaluated were synthetically generated from a number of distributions. These distributions included all the ones in the ANN library (Mount & Arya, 1997) (ver. 1.1), plus a few additional ones. The sets were generated for each run from the distributions using a different random number seed. For most distributions, the sets were generated using the ANN library itself, and were output as ARFF format of Weka for evaluation in Weka's Experimenter environment. The point distributions that were used for evaluation are listed in Table 4.4 and are discussed in brief below.

- **Uniform:** The points in this distribution were generated by drawing uniformly a

value from the interval [-1,1] for each of the dimensions, which, in the ANN library, is achieved through a uniform random number generator given by (Press et al., 1992). The points thus generated are uniformly distributed and do not exhibit any clustering or structure. This distribution, hence, formed the base case for the evaluated NN methods, to see how well they performed in the absence of any structure or cluster in the data.

- **Gaussian:** The points in this distribution were generated by drawing each coordinate (value of a dimension) of a point from the Gaussian distribution with mean 0 and standard deviation specified by the parameter `std_dev`. This distribution in the ANN library is implemented using Box, Muller, and Marsaglia's *polar method*, similar to the one described in (Knuth, 1997). For this thesis, the `std_dev` parameter for the generated sets was left to the default value of 1.0.

- **Laplacian:** In this distribution each coordinate $x_i$of a point was generated from Laplacian distribution with 0 mean and unit variance, using the following equation:

$$x_i = \frac{b}{2} \times e^{-b \times |x|},$$

where x is a uniform random variable in [0,1], and b was set to $\sqrt{2.0}$ to make the variance of the distribution equal to 1.0.

- **Correlated Gaussian & Correlated Laplacian:** These distributions model data from speech processing. For these distributions each coordinate x_i of points, for $i > 0$, was generated using the following recurrence relation:

$$x_i = \rho x_{i-1} + W_i,$$

where W_i is an independent and identically distributed random variable with mean 0, and ρ is the correlation coefficient parameter. The initial x_0 for the above equation is generated from the corresponding uncorrelated Gaussian or Laplacian distribution (using the above mentioned methods). For this thesis, ρ was set to 0.8.

- **Clustered Gaussian:** This distribution, given in the ANN library, is designed to model data that is clustered, and the clusters are full dimensional (i.e. they encompass all the dimensions to form their clusters). The distribution routine takes the number of clusters to generate as a parameter. It first randomly generates points as cluster centres (whose each coordinate is from [-1,1]) for the specified number of

clusters, and then uses the Gaussian distribution procedure mentioned above to generate points around each of the cluster centres with specified standard deviation `std_dev`. The points are generated evenly for all the clusters. For this thesis, the number of clusters was set to 4, and `std_dev` for the clusters was set to 0.1. Note, that in this and all the rest of the clustered distributions mentioned below, the cluster centres (and the specified `std_dev`) are same for both the data and the query set (for each run), allowing them to both have the same distribution.

- **Clustered Orthogonal Flats:** This distribution is designed to model points which are clustered on some m-dimensional axis-parallel flats/hyper-planes. It takes as parameters, the number of flats to generate, `max_clus_dim` as the maximum possible m for the flats, and `std_dev` as the standard deviation of the points along the *flat* dimensions. For each flat, the distribution routine randomly selects between 1 to `max_clus_dim` dimensions which would be wide, and treats the remaining unselected dimensions as flat dimensions. The points are then evenly generated for each flat by drawing their coordinates uniformly from [-1,1] for wide dimensions, and from Gaussian distribution, that has a randomly selected mean (from [-1.,1]) and `std_dev` standard deviation, for the flat dimensions. Thus, it generates points on hyper-planes for `std_dev=0`, and generates points that are slightly perturbed from the hyperplanes for `std_dev>0`, and each hyper-plane is at most `max_clus_dim`-dimensional (`max_clus_dim` $\leq d$) and is axis-parallel. For this thesis, number of flats was set to 3, `std_dev` was set to 0.1, and `max_clus_dim` was left to the default value of 1. This produced points on three 3-dimensional flats (planes) embedded in higher dimensions.

- **Clustered Ellipsoids:** This distribution is designed to model points which are clustered, and where each cluster (unlike the Clustered Gaussian above) lies close to a lower dimensional subspace. The distribution routine takes as parameters, the number of required clusters, `max_clus_dim`, `std_dev_lo` and `std_dev_hi`, and `std_dev`. It first randomly generates points as cluster centres for the require number of clusters, then for each cluster it randomly selects upto `max_clus_dim` number of dimensions and a standard deviation for each of the selected dimensions from the range `[std_dev_lo, std_dev_hi]`. Then, it generates points around each of the cluster centres drawing their coordinates from Gaussian distribution with the earlier selected standard deviation for selected dimensions, and with the `std_dev` standard deviation for unselected dimensions (mean of the Gaussian is the corresponding

coordinate of the cluster centre). This generates points in clusters that are elliptical in shape. For this thesis, number of clusters was set to 5, `max_clus_dim` was left to default 1, `std_dev_lo` was set to 0.01, `std_dev_hi` to 0.09, and `std_dev` to 0.1.

- **Straight Line:** This distribution was added to the ANN library. It generates points lying in a straight line which is embedded in high dimensions. This distribution was chosen because KDTrees are known to perform poorly on points in a straight line (Yianilos, 1993). The distribution routine generates each coordinate x_i, for $i > 0$, using the following slope-intercept equation:

$$x_i = mx_{i-1} + c,$$

and generates x_0 from the uniform distribution over [-1,1]. For this thesis, m and c were respectively set to 0.5 and 10.

- **Straight Line with Noise:** This distribution was also added to the ANN library. It was chosen to see how the NN methods behave if some perturbation/noise is present in points lying on a straight line. It generates points exactly as above (as in Straight Line distribution), and then simply adds some uniform noise to each coordinate of a point. The noise is added as follows:

$$x_i = x_i + \texttt{noiserate} \times \texttt{noisevariance} \times x,$$

where x is drawn from uniform distribution over [-1,1], and, for this thesis, `noiserate` and `noisevairance` were respectively set to 0.2 and 2.0.

- **Two Straight Lines:** This distribution was also added to ANN. It was chosen to see how the NN methods perform if more than one lines are present. It generates points in exactly the same way as mentioned above in Straight Line distribution. The only difference is that the second line has a slope of $-\frac{1}{m}$ and intercept $c/2$, and the points are generated such that there are even number of points among the two lines. The lines produced as such are non-parallel and non-intersecting.

- **Two Straight Lines with Noise:** This distribution also was added to ANN. It generates points exactly as above, and adds noise exactly as mentioned in Straight Line with Noise distribution above.

- **Plane:** The hyper-planes generated with Clustered Orthogonal Flats distribution, which is inbuilt in the ANN library, are axis-parallel. Hence, a distribution for non-

axis-parallel hyper-planes was also implemented. This distribution produces points on a single d-dimensional hyper-plane which is non-axis-parallel. The distribution routine generates the required d-dimensional points using the following equation, which is similar to the equation of a line in slope-intercept form:

$$x_{d-1} = mx_0 + mx_1 + \ldots + mx_{d-2} + c.$$

In the above, $x_0, x_1, \ldots, x_{d-2}$ are generated from uniform [-1,1], and, for this thesis, m and c were set respectively to 0.5 and 10.

- **Plane with Noise:** This distribution produces points exactly as above, except that extra noise is also added to the each of the coordinates of a generated point. The added noise (from uniform [-1,1]) is controlled by parameters `noiserate` and `noisevairance`, which were respectively set to 0.2 and 2.0.

- **Two Planes:** This distribution, also added to ANN, produces two hyperplane which are non-axis-parallel and are intersecting. It produces points by using $-\frac{1}{m}$ instead of m in the equation of the second plane and by ensuring that the number of points generated are even for the two planes.

- **Two Planes with Noise:** This distribution generates points exactly as above, but with added noise. The noise parameters `noiserate` and `noisevairance` were respectively set to 0.2 and 2.0.

- **Periodic Functions:** In all of the above distributions, the points are either in some form of clusters and/or on a line/plane. However, there can be situations where points lie on some form of a curve, which is a product of some function. Hence, to model such a scenario, which can occur in data from scientific/engineering observations, this distribution was implemented. It generates points lying on trigonometric functions. The distribution routine generates the points as follows. First the x_0 coordinate of a d-dimensional point is generated from uniform [-10,10], then the routine generates rest of the $d-1$ coordinates using $cos(x)$, $\frac{sin(|x|)}{|x|}$, and $sin(x)$ functions repeated in order on x_0, i.e. $x_1 = cos(x_0),\ x_2 = \frac{sin(|x_0|)}{|x_0|},\ x_3 = sin(x_0),\ x_4 = cos(x_0),$ ... and so on. The points generated as such lie on a single line that is curved in every dimension except the first. This distribution, unlike the others, was not implemented in ANN, instead it was implemented as a separate class under Weka.

- **Periodic Functions with Noise:** This distribution generates the points exactly as above, and then simply adds uniform noise to the points. Unlike the other noisy distributions, the noise in this distribution is added from uniform [0,1] and is only controlled by the `noiserate` parameter. The `noisevairance` parameter is hard coded to be the range of the dimension to which the noise is being added, which makes the `noiserate` parameter more meaningful. For this thesis, the `noiserate` parameter was set like the other distributions to 0.2, whereas the distribution was implemented like the above as a separate class under Weka.

All of the above distributions are illustrated graphically in 2 dimensions in Figure 4.1.

4.3 Results

The representative results of the experimental evaluation of the NN methods are presented in the following subsections. First results for the construction methods of KDTrees are presented, then the results for the construction methods of Metric Trees. Finally results comparing the different NN methods are presented. The subsections also present a discussion and analysis of their respective results.

The preprocessing/construction performance of the methods is assessed in terms of the CPU preprocess time. Whereas, the query performance of the methods is assessed in terms of CPU query time per query, and Avg. Points Visited per query for the given query set (these measures are described above in Section 4.1). Avg. Points Visited is used in addition to the CPU query time because it is a more direct measure of a method's effectiveness as all the evaluated methods achieve speed up by trying to reduce the number of points inspected during kNN search. However, the additional overhead involved in reducing the number of points inspected entails the assessment of methods' CPU query time as well.

4.3.1 KDTrees' Construction Methods

Let us first look at the construction times of the selected construction methods of KDTrees. Figure 4.2 shows the plot matrix of construction time of the methods for increasing data size (n) for $d = 16$, and Figure 4.3 shows the plot matrix of construction time for increasing d for $n = 100000$. In both the figures the matrices contain plots for each of the evaluated point distributions, and all the plots have their x and y axes on log scale. This basic pattern of figures for presenting results is also used for all the rest of the results that are presented later in the thesis.

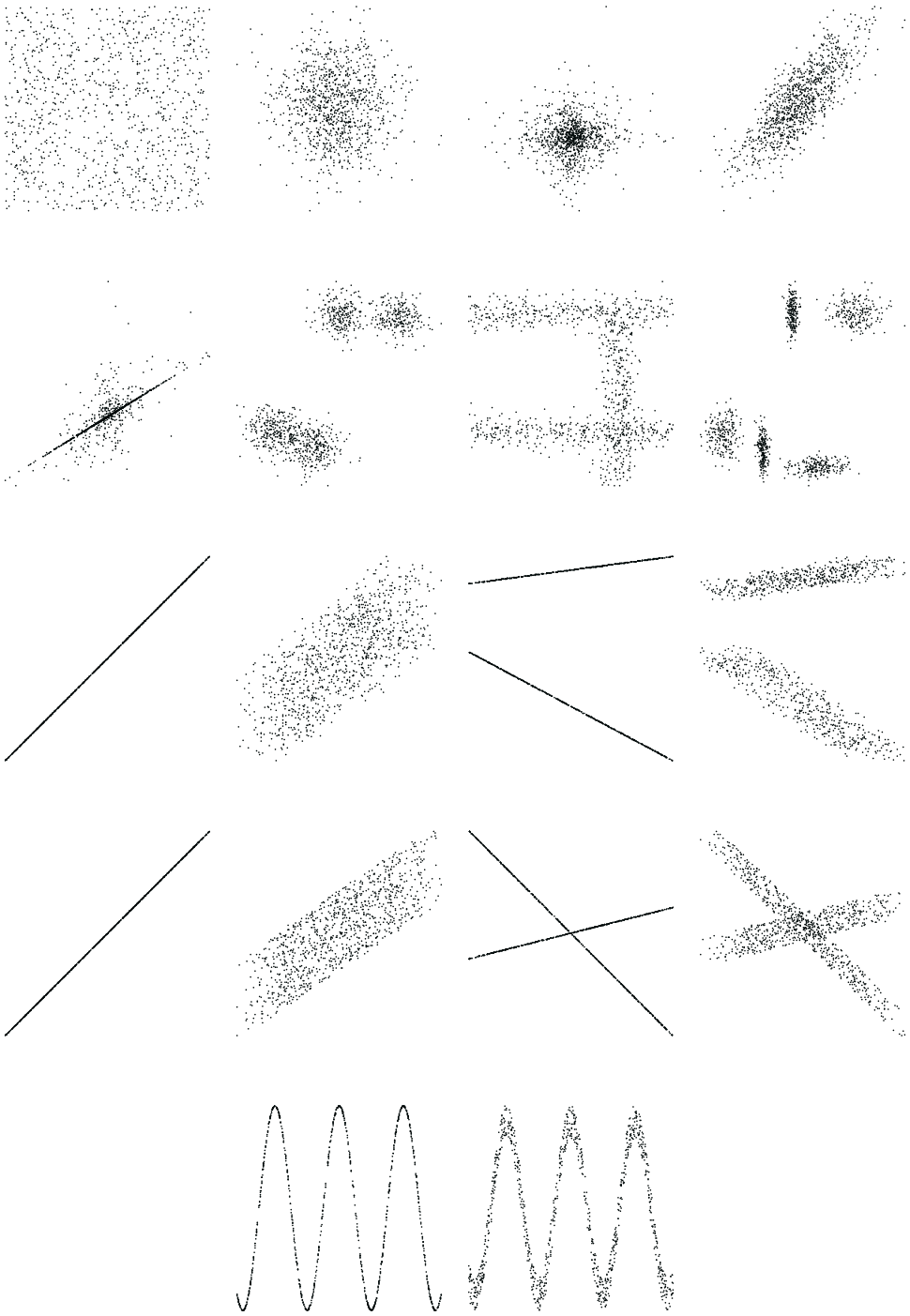


Figure 4.1: Evaluated Point Distributions (from left to right and top to bottom): (1) Uniform, (2) Gaussian, (3) Laplacian, (4) Correlated Gaussian, (5) Correlated Laplacian, (6) Clustered Gaussian, (7) Clustered Orthogonal Flats, (8) Clustered Ellipsoids, (9) Straight Line, (10) Straight Line with Noise, (11) Two Straight Lines, (12) Two Straight Lines with Noise, (13) Plane, (14) Plane with Noise, (15) Two Planes, (16) Two Planes with Noise, (17) Periodic Functions, and (18) Periodic Functions with Noise.

In Figure 4.2 for comparison the $n\log n$ line is also plotted, as all the KDTree construction methods that are evaluated work in $O(n\log n)$ time. Since the values on y-axis in the figure represent CPU time, which are much less than the actual values of $n\log n$, the $n\log n$ line has been shifted down to lie with the rest of lines for easier comparison. It can be noticed in the figure that for all the methods, on (almost) all the point distributions, as n increases the construction time stabilizes and grows at a constant polynomial rate. Also, the slope of each method's construction line is slightly higher than that of the $n\log n$ line, suggesting that the construction time of each method grows at a rate which is polynomial but slightly higher than $n\log n$.

Dependence of the construction time of the methods on d is shown in Figure 4.3. For comparison, the figure also includes a shifted $y = d$ line . It can be noticed from the figure that for all the methods the construction time grows at a slightly increasing exponential rate. However, in almost all cases all the methods, with the exception of KMeans and SlMidPt, grow sublinear in d (i.e. at a rate $d^{x<1}$), since the slope of their lines is less than that of the line $y = d$. KMeans grows at a rate higher than d, as it is known to work in $O(d^2)$. SlMidPt, which does not have any runtime guarantees, as can be noticed in the figure, almost always grows less than or equal to d except on the distributions where points lie on a line (i.e. the straight line and functions distributions). KMeans empirically, in plots not shown here in the main text, seems to grow at a rate more similar to $d^{1.5}$ rather than the theoretical d^2, whereas the SlMidPt method always grows slower than $d^{1.5}$ except on the line distributions where it approaches $d^{1.5}$.

Overall from both the figures it can be seen that KMeans has the highest construction time, Median and MidPt methods in every case are similar, and SlMidPt is higher than Median and MidPt but only on clustered and line distributions and only for higher dimensions. This higher cost of SlMidPt is probably due to its trade-off of better fit to the structure of the data. Trends similar to what has been described, were also observed in plots similar to Figure 4.2 and Figure 4.3 but with different fixed values of n and d. This can be observed in another two plots similar to Figure 4.2 and Figure 4.3 but with different fixed values of n and d that have been placed in Appendix A. Furthermore, in Appendix A Figure 4.3 plot with $y = d^{1.5}$ line instead of $y = d$ is also included to illustrate the empirically observed growth rates of KMeans and SlMidPoint methods.

Let us now look at the query performance of the selected construction methods of KDTrees. Figure 4.4 for $k = 5$ and $d = 4$, shows the average points visited by the methods for increasing n on non-uniform query (i.e. having the same distribution as that of the data), and Figure 4.5 for $k = 5$ and $d = 4$, shows the average points visited for increasing

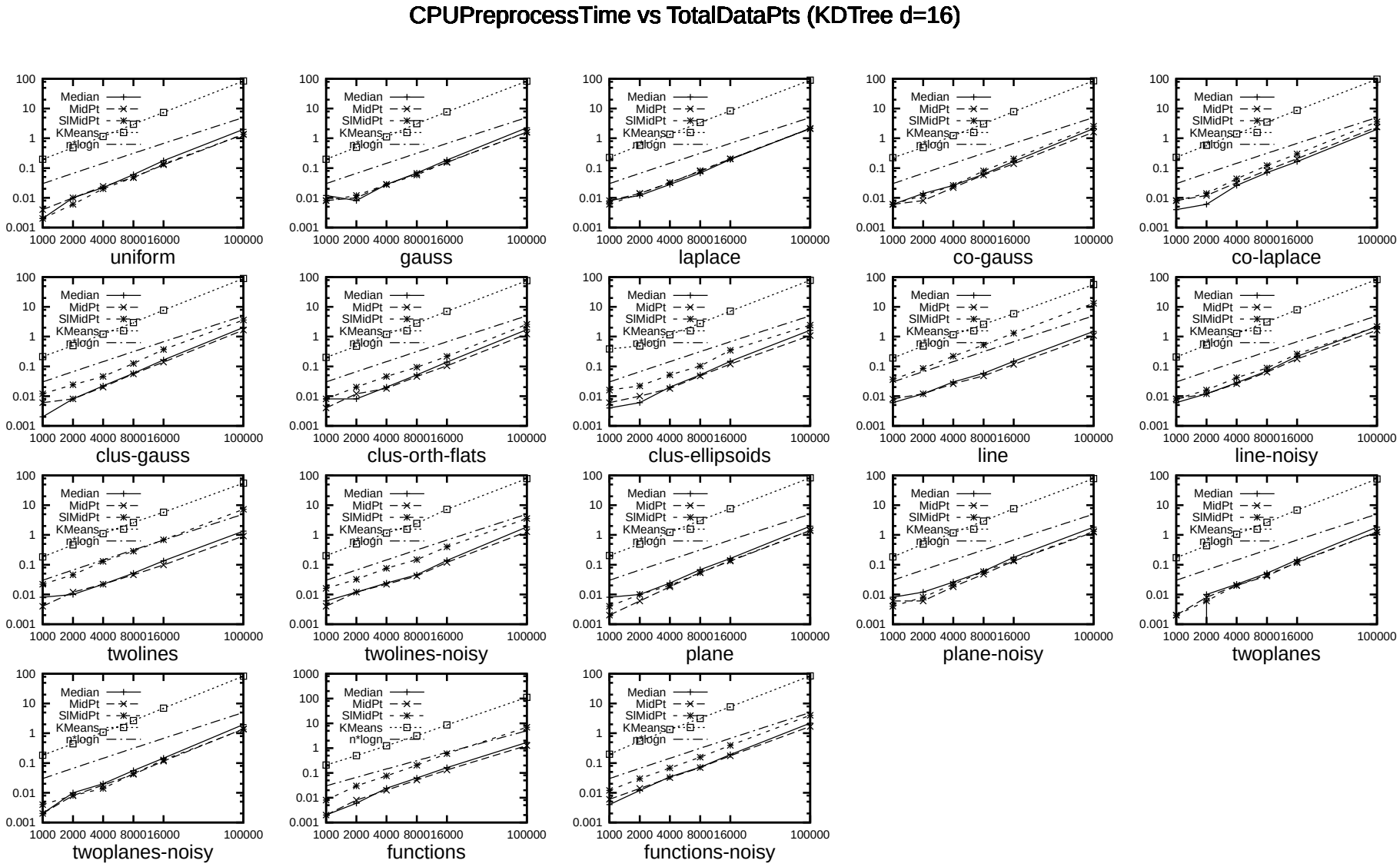


Figure 4.2: KDTrees' construction time for increasing n.

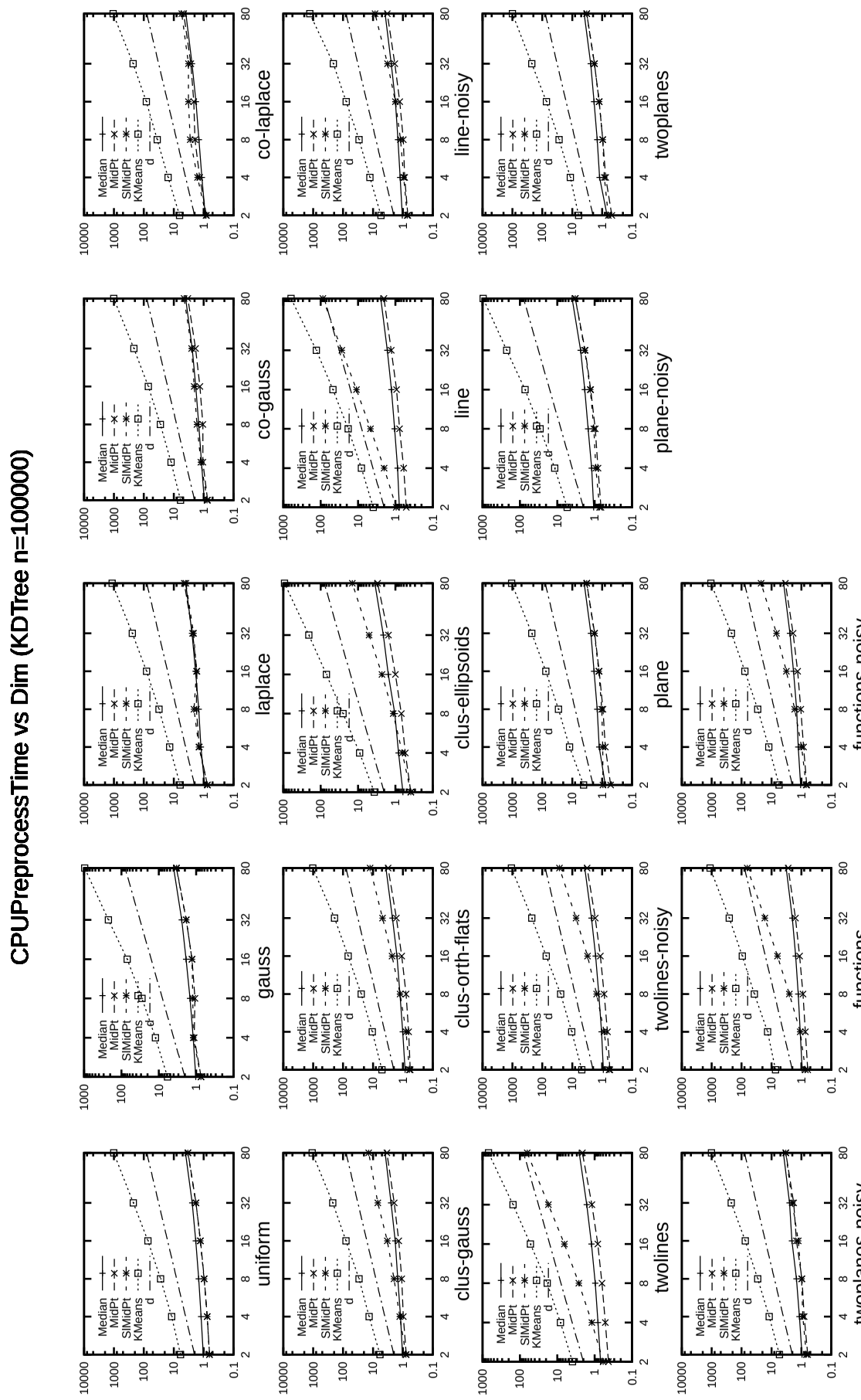


Figure 4.3: KDTrees' construction time for increasing d.

n on uniform query. Theory tells us that KDTrees in expected case should inspect log$n$ points. In both figures, 4.4 and 4.5, log$n$ line is also plotted for comparison. It can be seen in case of non-uniform query in Figure 4.4, that apart from co-laplace, clus-gauss and non-noisy line distributions, all the methods in the end grow at a rate slower or equal to log$n$. Whereas, in case of uniform query in Figure 4.5, on almost all distributions (all but uniform), all methods apart from SlMidPt from the very start grow at a rate faster than log$n$. SlMidPt in case of uniform query grows at or near the rate of log$n$ on all but the non-noisy line distributions. KDTrees are known to perform poorly on points lying on a line, and this is evident from the exponential growth of every construction method on non-noisy line distributions in case of non-uniform query (when query points are also on the same line), and in case of uniform query, from SlMidPt's exponential growth near the end, on non-noisy line distributions. All the rest of the exceptions in which the rate of growth of one or more methods is greater than log$n$, both in case of uniform and non-uniform query, the rate actually approaches the rate of linear growth n. However, in every exception the number of average points visited is still less than n. This can be seen in figures similar to 4.4 to 4.5 in Appendix A which are plotted with $y = n$ line. Hence, from the figures (4.4 & 4.5) it seems the log$n$ rate of growth can only be achieved by KDTrees on non-uniform query (when the query follows the distribution of the data), and in case of uniform query, only with SlMidPt construction method. It ought to be mentioned that only the rate of growth of log$n$ is achievable, as in both figures 4.4 and 4.5 the log$n$ line has been shifted up and the actual number of average points visited is some constant times higher. The actual number is though still sublinear, which can be seen in the equivalent figures just mentioned that are placed in Appendix A with (unshifted) $y = n$ line.

The trends in Figure 4.4 and Figure 4.5 become much clearer when we consider figures 4.6 and 4.7 which show the average points visited for increasing d for $n = 100000$ and $k = 5$, for respectively non-uniform and uniform query. In Figure 4.6, for most distributions a peculiar S shape can be noticed, which is not so present in Figure 4.7, especially not for methods other than SlMidPt. This is because of log$n$ growth rate, that is only possible for KDTrees on lower d's, which then grows exponentially and tapers off at the polynomial rate n as d becomes large. Since, for uniform query all the construction methods apart from SlMidPt grow nearer to the (polynomial) linear rate n even for lower dimensions, this peculiar S shape is not so present in Figure 4.7, and the number of points visited in most cases for these methods grows at a constant (polynomial) rate with respect to d, before tapering off at linear rate n. This, thus, further reasserts the fact for log$n$ growth that is observed in figures 4.4 and 4.5 above, and also further shows, in line with the theory,

that the log$n$ growth rate is only achievable by KDTrees for lower d's. As d becomes large the trees tend to degenerate to simple linear search. This can be observed from both the figures, where the trees either degenerate completely to simple linear search, visiting all the 100000 points, or taper off at the rate n. However, this does not apply to non-noisy line distributions, which stand out as an exceptional case. As can be seen in Figure 4.6 for non-uniform query, the number of points visited is almost always constant for non-noisy line distributions. Only on functions distribution for KMeans it grows, but still slower than the exponential rate, while for SlMidPt on straight line distributions it grows in the negative direction. This shows that for non-noisy line distributions the number of points visited grows exponentially (while still being sublinear) only with respect to n, while it stays almost constant for increasing d's. This, however, only holds true for non-uniform query, and, as can be seen in Figure 4.7, not for uniform query (when the query points are not on the line). Another peculiar feature that ought to be pointed out in Figure 4.6, is that for co-laplace and planar distributions, the number of points visited for $d = 4$ is less than that for $d = 2$. This is because in $d = 2$ these distributions form a line, as can be seen in Figure 4.1, and hence, because of the exponential rather than the logarithmic rate of growth the number of points visited for these distributions is higher at $d = 2$ (especially with the high value of $n = 100000$ selected for the figure). A figure similar to 4.4 but with $d = 2$ is placed in Appendix A to show the exponential growth on these distributions at $d = 2$.

Overall in summary, figures 4.4, 4.5, 4.6 and 4.7 indicate that all the construction methods of KDTrees apart from SlMidPt offer logarithmic rate of growth (O(log$n$), as it grows similar to log$n$ but is some constant times higher) only for lower d's and only for non-uniform query, which grows exponentially towards linear n as d gets higher (see Figure A.4 similar to Figure 4.4 but for $d = 32$). SlMidPt offers similar growth but is not restricted to just non-uniform query, it offers it for both non-uniform as well as uniform query. The methods other than the SlMidPt on uniform query give near linear rate of growth (near to O(n) but strictly less than n), which also grows towards linear n as d gets higher. All the construction methods including SlMidtPt grow exponentially (though still being sublinear) with n for non-noisy line distributions, while remain constant for increasing d. However, the methods apart from SlMidPt, only exhibit this for non-uniform query (when query points are also on the line), and when the query is uniform even at moderate d's they degrade to linear n . SlMidPt on non-noisy line distributions with uniform query, grows exponentially with d, but even at higher d's stays significantly sublinear. Trends similar to what has been described were observed with fixed values of k, d and n other than the

ones used in figures 4.4, 4.5, 4.6 and 4.7, and hence the figures presented represent the overall trends in the complete results.

In terms of average points visited, as can be seen from figures 4.4, 4.5, 4.6 and 4.7, SlMidPt construction method of KDTrees is superior to other methods on clustered and line distributions when the query is uniform, whereas when the query is non-uniform it behaves almost exactly (infact better on line distributions) as the other methods. Though, it does exhibit higher construction time ($d^{1.5}$) on line distributions but this comes at a trade-off of better fit to the data which, as can be seen from the figures, gives better query performance. This behaviour of the SlMidPt method also holds in terms of CPU query time. This can be seen in figures equivalent to 4.4, 4.5, 4.6 and 4.7, placed in Appendix A, which plot CPU query time rather than average points visited for increasing n and d. Hence, the SlMidPt construction method was selected for KDTrees, for their further comparison with the rest of the NN methods.

4.3.2 Metric Trees' Construction Methods

Let us now look at the selected construction methods of Metric Trees. Figure 4.8 shows the construction time of the methods for increasing n for $d = 16$, and Figure 4.9 shows the construction time for increasing d for $n = 16000$. It can be seen in Figure 4.8 that the construction time of the MiddleOut method grows at a rate which is slightly exponential, while the rest of the methods grow at a rate which is a (fairly) constant polynomial but slightly higher than $n\log n$. While MiddleOut and PointsClosestToFurtherPair do not have any guarantee, the rest of the methods in theory work in O($n\log n$). With respect to d, it can be seen in Figure 4.9, that the methods grow at a variable rate which is either lower or near to linear d. Overall from both the figures, it can be seen that MedianValue and MedianDistance have very similar construction times. FurthestPair exhibits growth pattern similar to the two, but is a constant times higher. Whereas, MiddleOut has the highest construction time and does not follow the growth pattern of the rest of the methods. Similar trends to what has been described were observed in plots for other fixed values of n and d. Two more similar plots with different values of n and d can be found in Appendix B.

Let us now look at the methods' query performance. Figures 4.10[1] and 4.11[2] show respectively for non-uniform and uniform query, the average points visited by the methods for increasing n, for $d = 4$ and $k = 5$. It can be seen in Figure 4.10[1] that for non-uniform query all the methods apart from MedianDistance, in the end have a growth rate lower or near (slightly higher) to $\log n$. For uniform query, these methods (other than

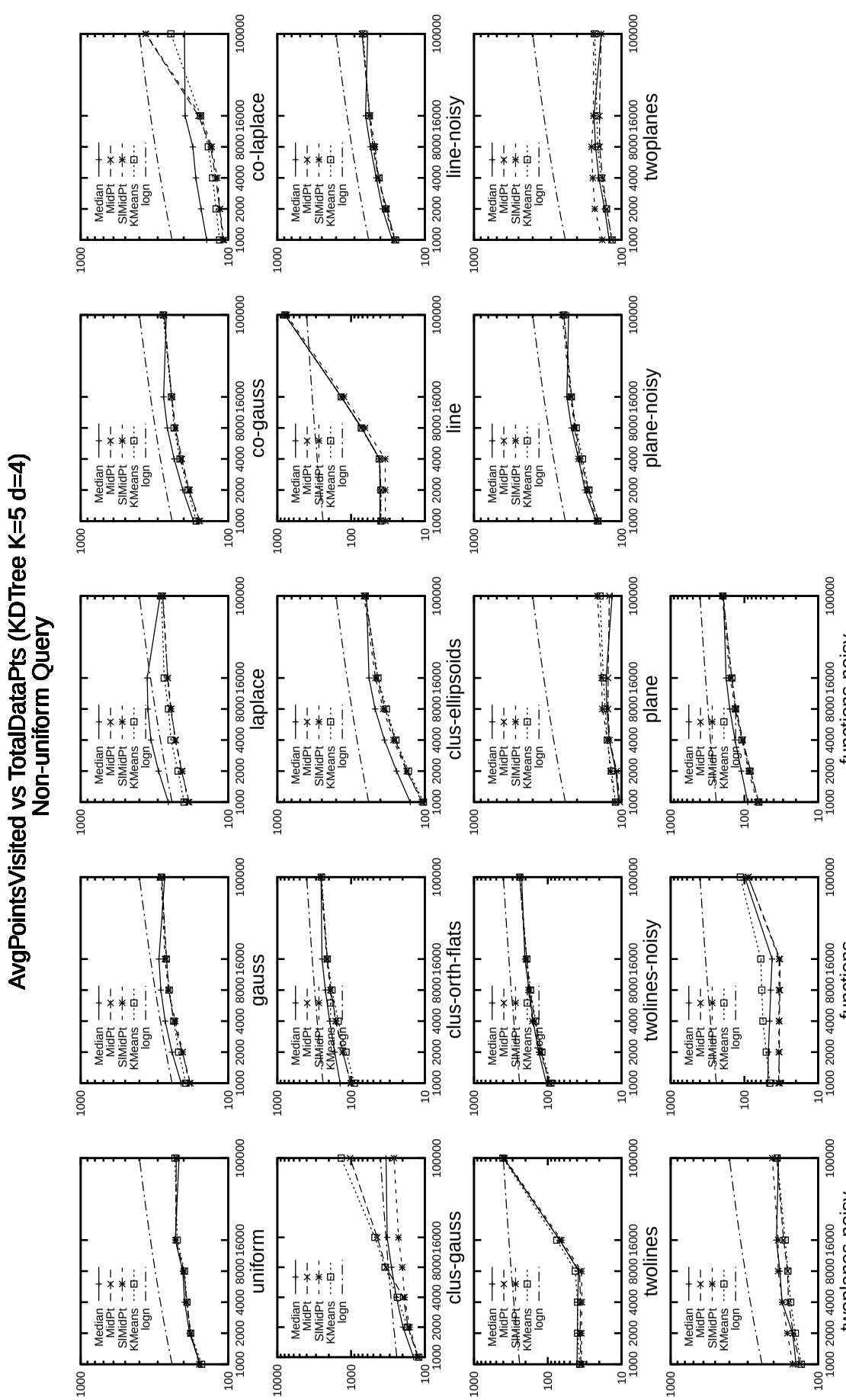


Figure 4.4: Avg query points visited by KDTrees for increasing n on non-uniform query.

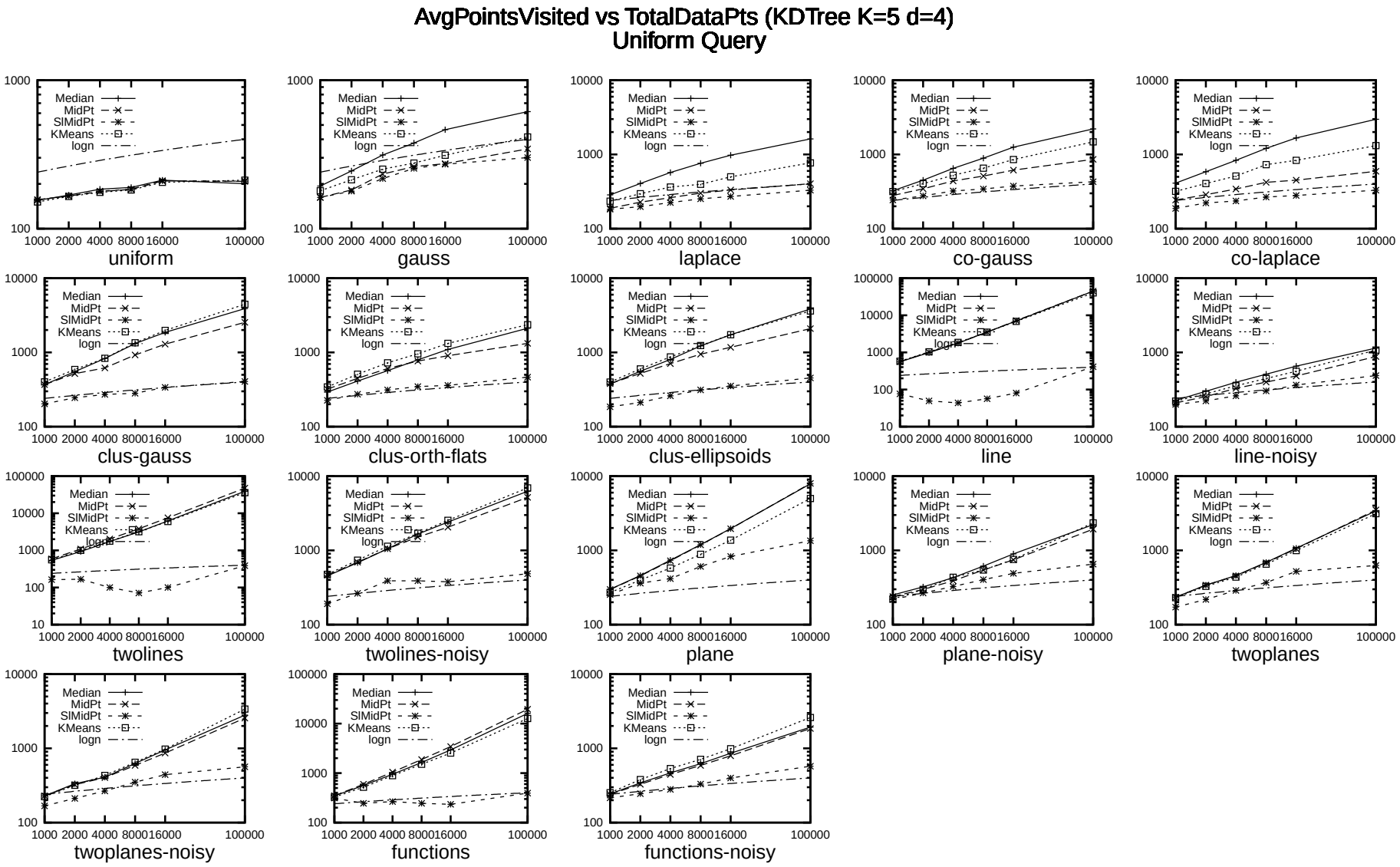


Figure 4.5: Avg query points visited by KDTrees for increasing n on uniform query.

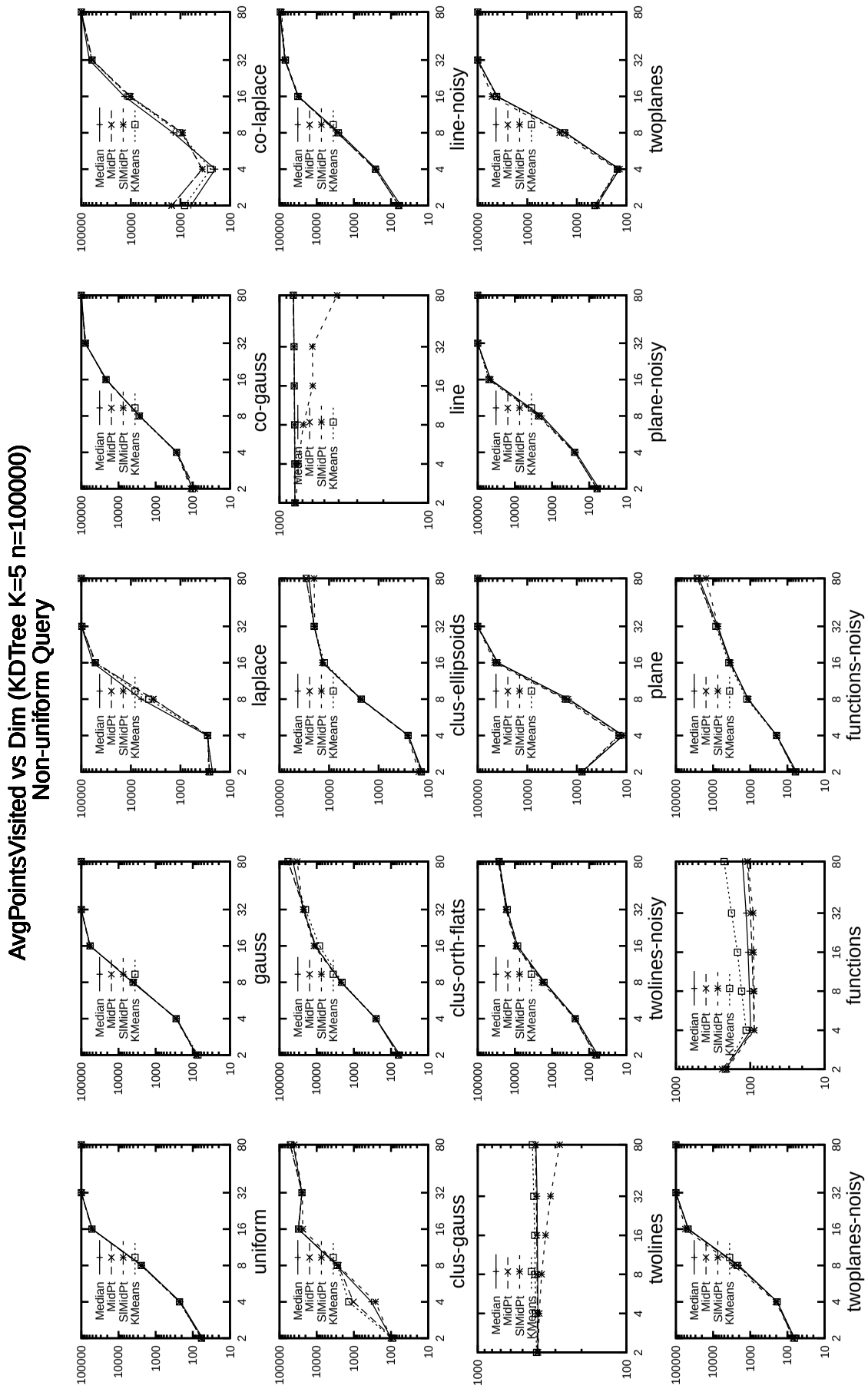


Figure 4.6: Avg query points visited by KDTrees for increasing d on non-uniform query.

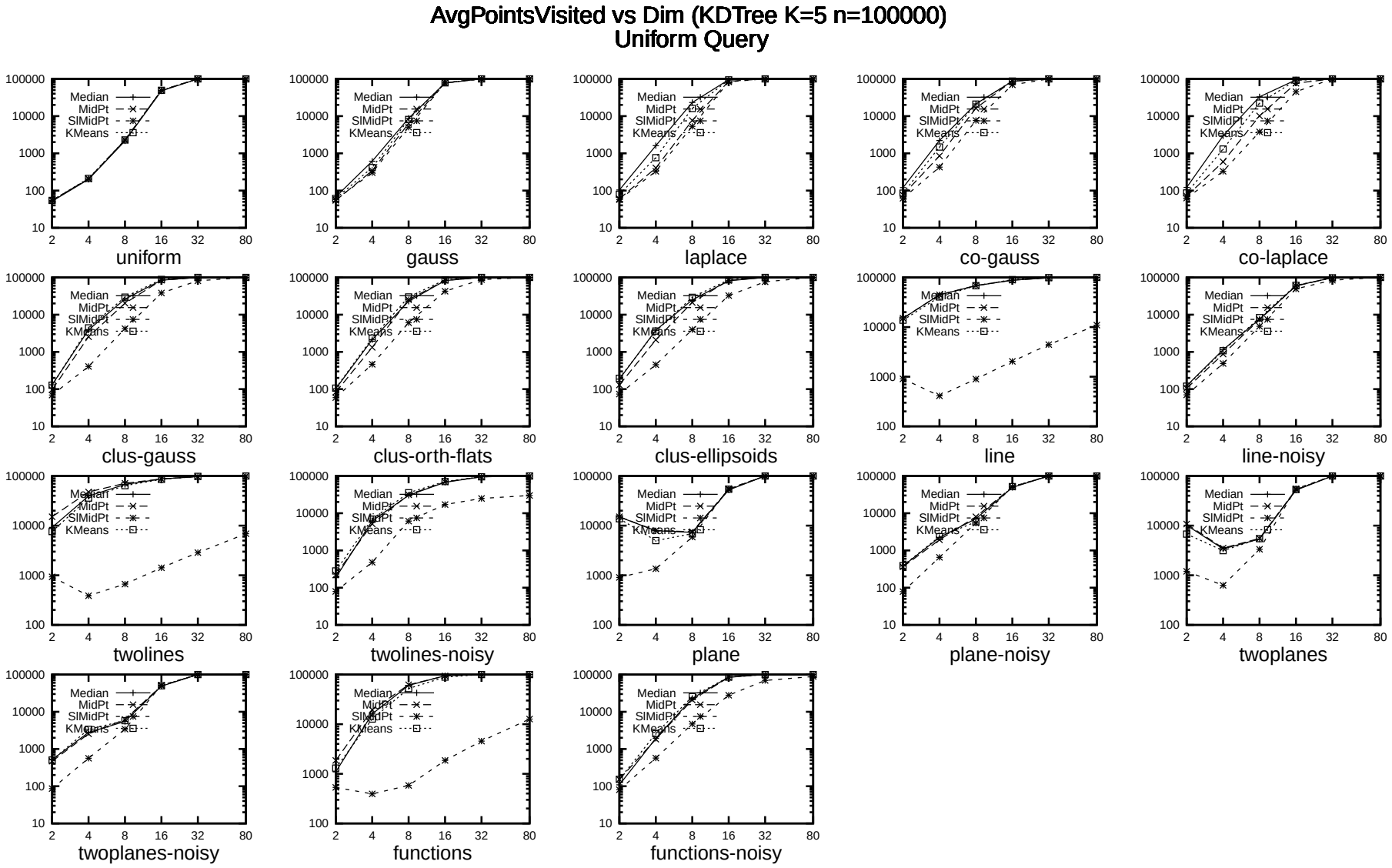


Figure 4.7: Avg query points visited by KDTrees for increasing d on uniform query.

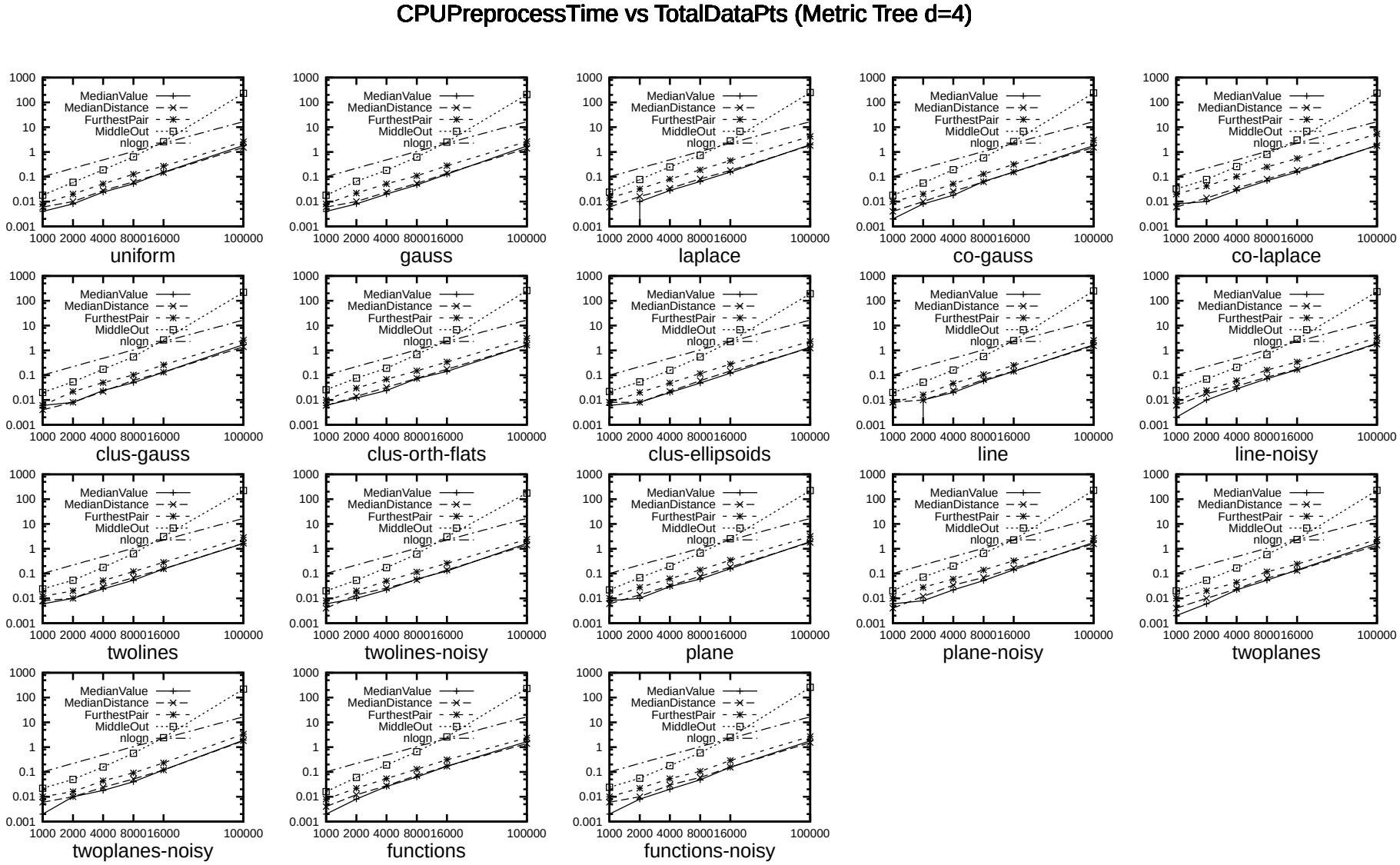


Figure 4.8: Metric Trees' construction time for increasing n.

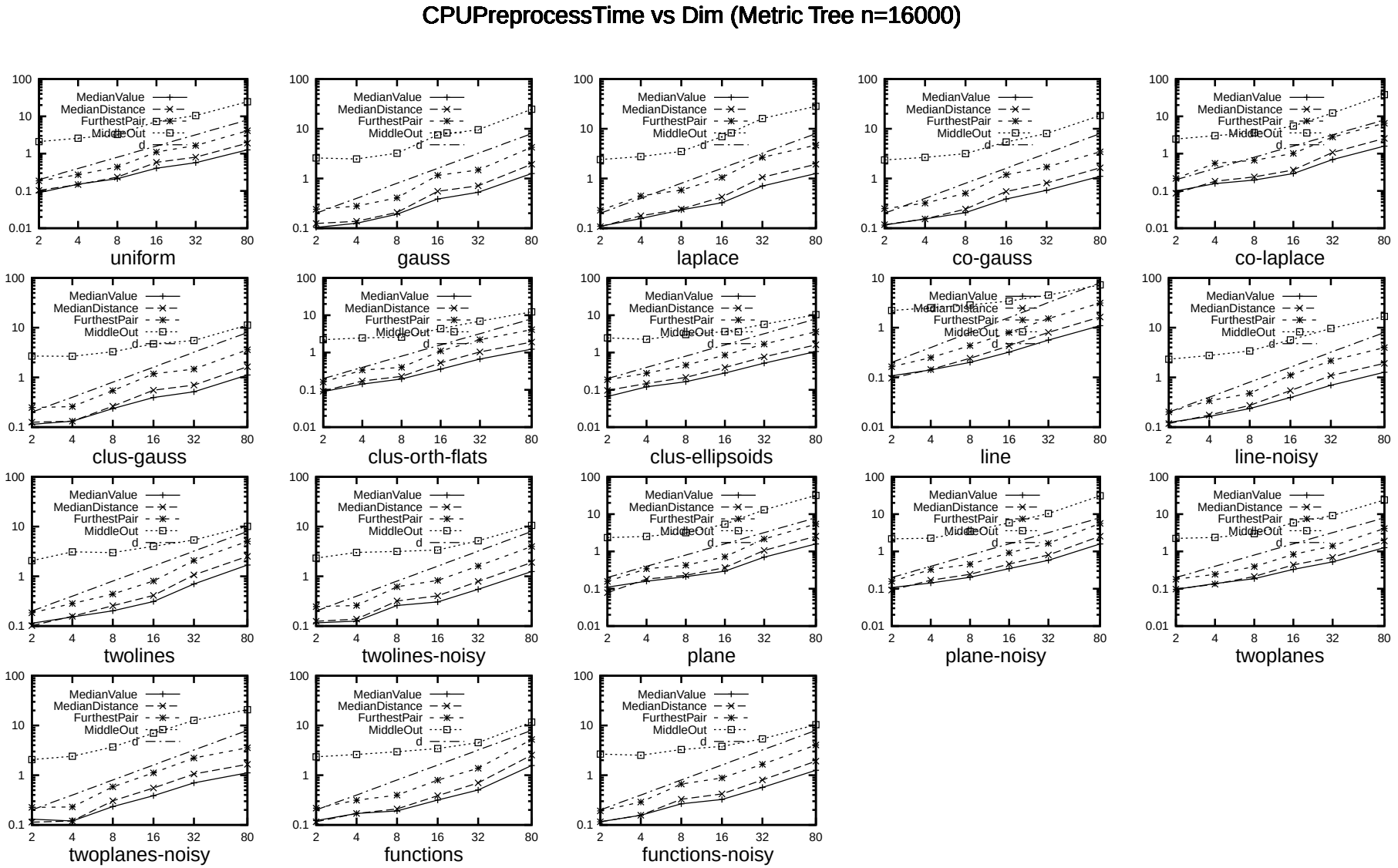


Figure 4.9: Metric Trees' construction time for increasing d.

MedianDistance) show mixed trends. On non-noisy line and planar distributions their rate gets higher, whereas on clustered and co-laplace distributions FurthestPair and MiddleOut get slightly lower but MedianValue does not show any clear trend. In all the rest of the cases also there are no clear trends for these three methods. The MedianDistance method for both non-uniform and uniform query has growth rate which is consistently higher than log$n$. The number of points visited by MedianDistance is also consistently higher than all the rest of the methods, except in the case of laplace distribution where it gets closer to MedianValue for both non-uniform and uniform query. It ought to be emphasized all the methods only have log$n$ rate of growth, the actual number of points visited is a constant times higher.

Figures 4.12[3] and 4.13[4], for respectively non-uniform and uniform query, plot the average points visited for increasing d for $n = 100000$. In both the figures it can be seen that all the methods, like the methods of KDTrees, grow quickly with respect to d (except on non-noisy line distributions) and taper off at linear rate n. MedianValue is the worst overall. All the rest of the methods mostly grow at a constant polynomial rate (non-exponential) before tapering off at linear rate O(n). However, in some cases, especially on planar distributions, one or more of these methods also show the peculiar S shape like the KDTree methods, suggesting in those cases their logarithmic growth for lower dimensions. The non-noisy straight line distributions are also an exception, where in the end all the methods remain either constant or grow in the negative direction with respect to d for both non-uniform and uniform query. Some methods on non-uniform query also exhibit this for non-noisy functions distribution (non-noisy unstraight line), but on uniform query they (and the rest of the methods) do not stay constant but grow at a slow rate. Similar trends were also observed in figures similar to 4.10, 4.11, 4.12 and 4.13, but with different fixed values of k, d and n.

From all the four figures, 4.10, 4.11, 4.12 and 4.13, it can be seen that MedianDistance is the worst method overall (its better than MedianValue only for smaller d, and only on laplace distribution). However, as can be noticed from the figures, there is no candidate among the remaining methods which consistently (or almost consistently) performs better than the others. All the remaining three methods are either sometimes better or sometimes worse compared to each other. This indeed posed a dilemma for selecting a construction method for further comparison of the trees with other NN methods. A closer inspection of the figures, however, reveals that the FurthestPair method almost always ends up near the bottom, and (almost always) is either the best or the second best in terms of query performance. This also holds true if we look at CPU query time rather than average points

visited, as can be seen in figures B.3, B.4, B.5 and B.6 in Appendix B that are respectively similar to figures 4.10[1], 4.11[2], 4.12[3] and 4.13[4]. The construction time of FurthestPair is also on par with the method best in terms of construction time (MedianValue), while still being significantly lower than the one which is the worst (MiddleOut). Hence, the FurthestPair method was selected for Metric Trees, for their further comparison with other NN methods.

4.3.3 NN Methods

Let us now look at the best construction methods of KDTrees and Metric Trees compared with the other NN methods, i.e with the Annulus Method, Cover Trees and simple Linear Search (with PDS). Since Linear Search does not perform any preprocessing, comparing its CPU preprocessing time with the others would not serve any purpose, therefore it is compared with the others only in terms of its query performance.

Figure 4.14 plots the preprocessing/construction time of the methods (other than Linear Search) for increasing n for $d = 4$, and Figure 4.15 plots the preprocessing/construction time for increasing d for $n = 100000$. It can be seen in Figure 4.14 that apart from Annulus Method all the rest of the methods have preprocessing times which (mostly) grow at a rate slightly higher than $n\log n$. Whereas, Annulus Method in all cases has a growth rate lower than $n\log n$, which is because of its lower $O(n)$ preprocessing cost. With respect to d, it can be seen in Figure 4.15, that apart from Cover Trees all methods in almost all cases grow at a rate either lower or equal to d. Cover Trees curiously grow at a rate lower than d only upto $d \leq 8$, after which for all but the non-noisy line distributions their construction time shoots up exponentially. On non-noisy line distributions, though, they grow slower than d even at higher d's. This peculiar behaviour of Cover Trees is not fully understood. Literature states their construction time to be $O(c^6 n\log n)$, and it could be that the expansion constant c becomes quite large after $d = 8$ for all but non-noisy line distributions. However this can not be fully ascertained, as the literature does not state explicitly either the Cover Trees' or the expansion constant's dependence directly on d.

Overall from figures 4.14 and 4.15, it can be seen that Annulus Method has the lowest preprocessing time. SlMidPt of KDTrees is the next best method; however, on non-noisy line distributions it becomes the worst for higher d's. Cover Tree is similar to FurthestPair upto $d \leq 8$, after which it becomes the worst method on all but the non-noisy line distributions. On non-noisy line distribution it is the worst method for lower d's but becomes the best for higher d's. Metric Trees' FurthestPair method is generally in between SlMidPt and Cover Tree, either worse than KDTree and better than Cover Tree,

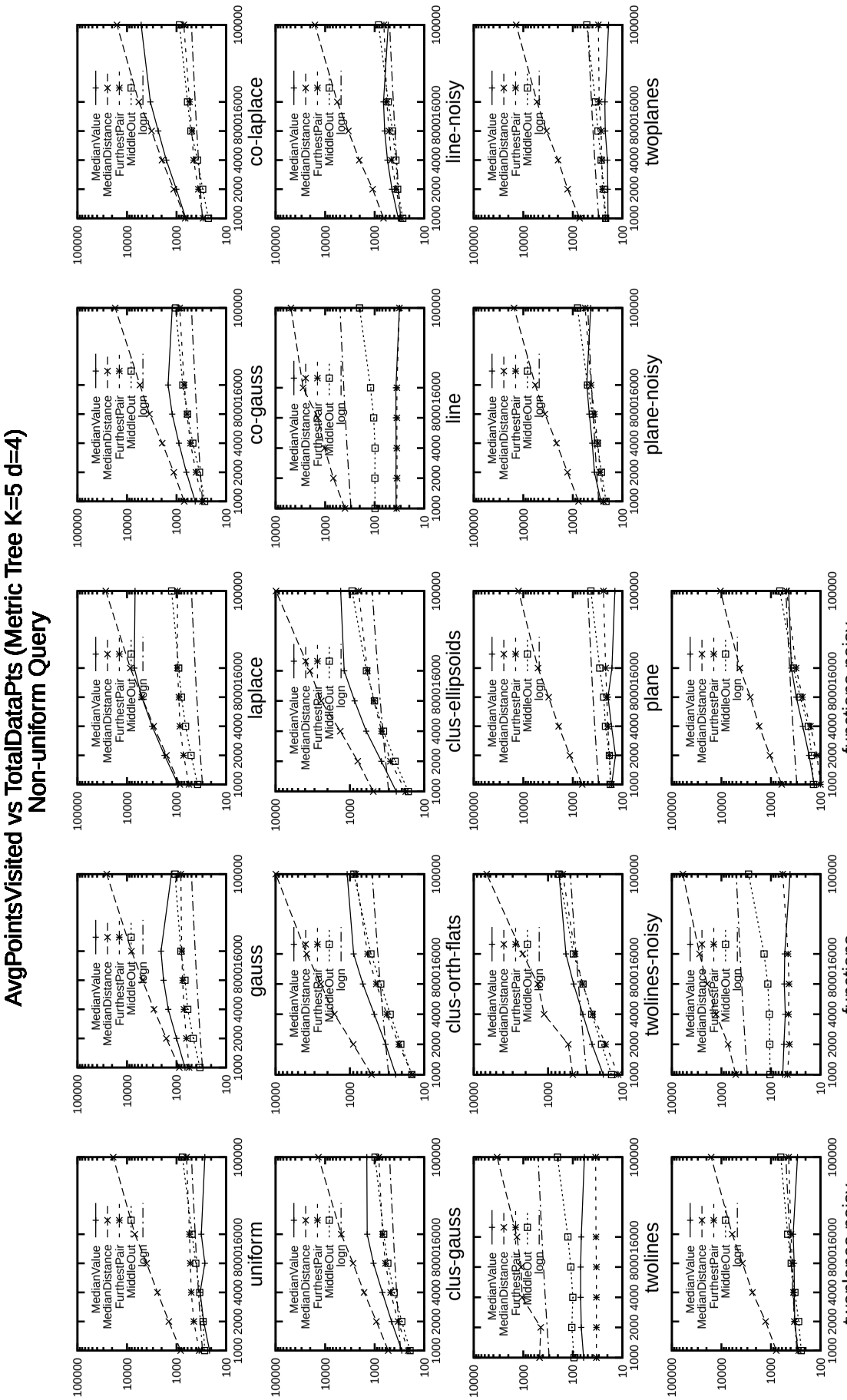


Figure 4.10: Avg points visited by Metric Trees for increasing n on non-uniform query.

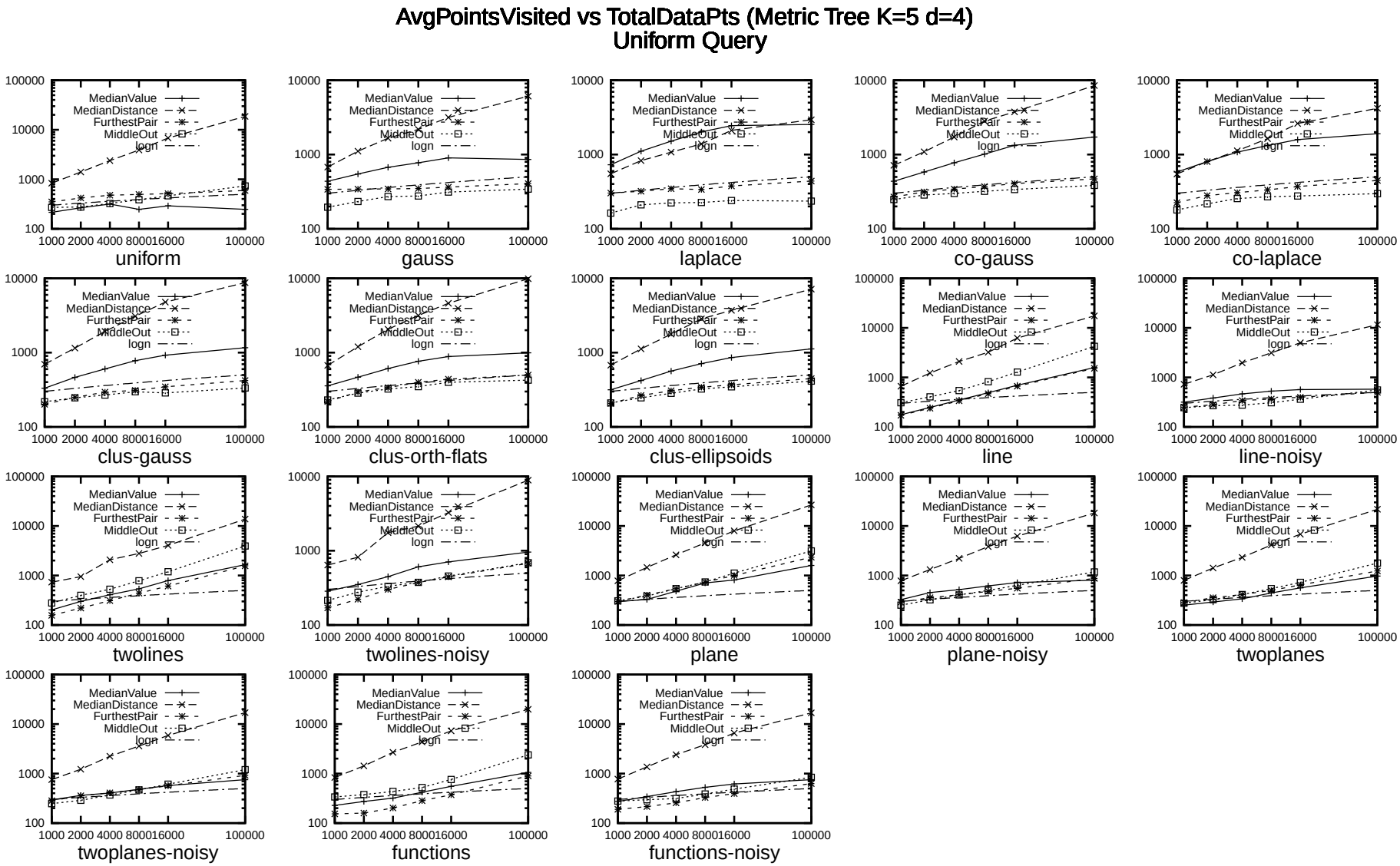


Figure 4.11: Avg points visited by Metric Trees for increasing n on uniform query.

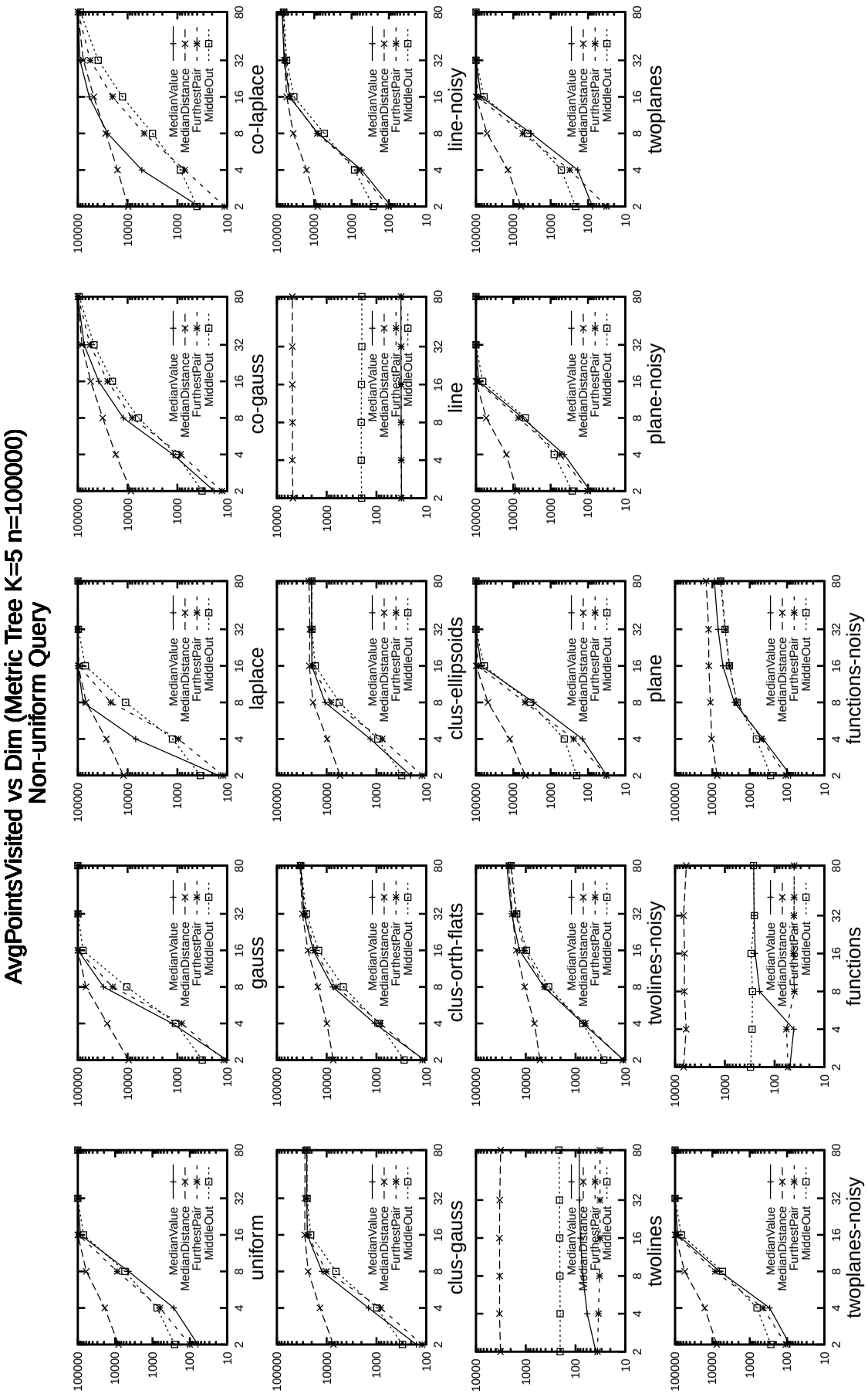


Figure 4.12: Avg points visited by Metric Trees for increasing d on non-uniform query.

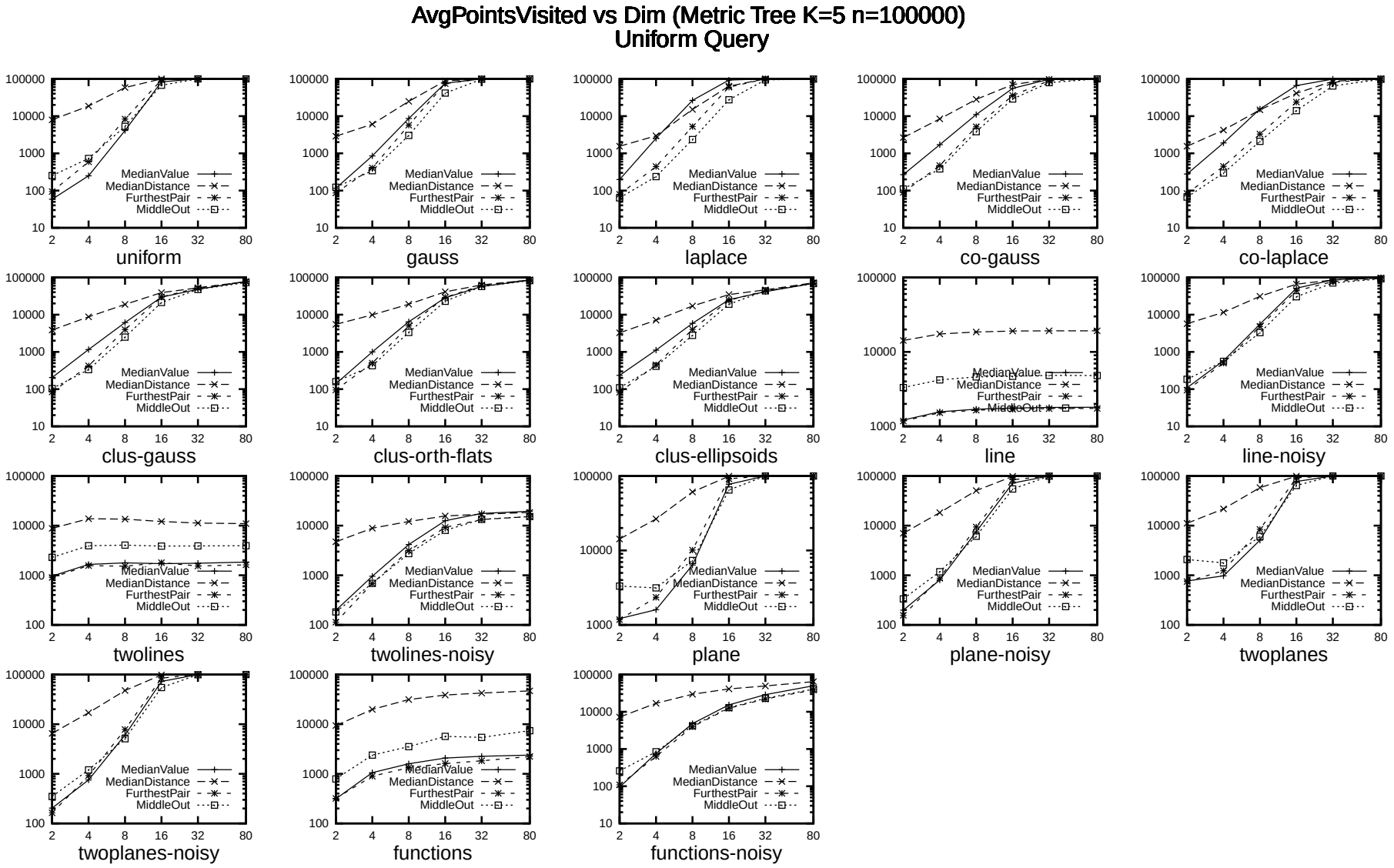


Figure 4.13: Avg points visited by Metric Trees for increasing d on uniform query.

or vice versa, in case of non-noisy line distributions. The only notable exceptions to this are the co-laplace and functions distributions, where FurthestPair is the worst method for smaller d's (< 16). Two more figures similar to 4.14 and 4.15 but with different fixed values of d and n can be found in Appendix C.

Let us now look at the query performance of the methods. Figures 4.16[1] and 4.17[2] for $k = 5$ and $d = 4$, plot the average data points visited by the methods for increasing n for respectively non-uniform and uniform query. It can be seen for uniform query in Figure 4.16[1], that the Annulus Method has the highest growth rate and is also the worst method in most of the cases. Only on non-noisy line distributions it performs better and is either the best or one of the best methods (low d's on co-laplace distribution is also a notable exception, where it performs better). On distributions other than non-noisy lines, KDTree is the best method. On non-noisy line distributions it grows exponentially, and for non-noisy unstraight lines it is the worst method for higher n's. Metric Tree has a growth rate similar to KDTree but is a constant times worse. Only for higher n's on non-noisy line distributions it becomes better than KDTree. Cover Tree has a higher growth rate than both KDTree and Metric Tree, on all but the non-noisy line distributions. On non-noisy line distributions its growth is similar to Metric Tree but lower than KDTree at higher n's. It is also the worst method on non-noisy line distributions at lower n's compared to KDTree and Metric Tree, at higher n's KDTree either becomes the worst or for unstraight line (functions distribution) comes close to Cover Tree. On rest of the distributions (apart from non-noisy lines) Cover Tree is mostly in the middle, worse than KDTree but better than Metric Tree, especially for $n < 100000$. In Figure 4.17[2] for uniform query it can be seen that Annulus Method again has the highest growth rate and it is the worst method overall compared to others. On most distributions its rate of growth and the actual number of point visited get much closer to that of Linear Search. Cover Tree is the best method or one of the best methods on all but the non-noisy straight line distributions[1] for $n < 100000$. Its rate of growth also gets much closer to KDTree and Metric Tree, compared to its rate in case of non-uniform query. On non-noisy straight line distributions KDTree is better than Cover Tree and is the best method, even though on these distributions it exhibits a much higher rate of growth than either Cover Tree or Metric Tree. On all the rest of the distributions (other than non-noisy straight lines) KDTree and Metric Tree are mostly similar, and mostly worse than the Cover Tree. Only in some cases one is better or worse than the other, or better than the Cover Tree for

[1]Cover Tree is also not the best on Uniform distribution, but the case of uniform distribution is exactly the same as in Figure 4.16[1] and hence is not considered here.

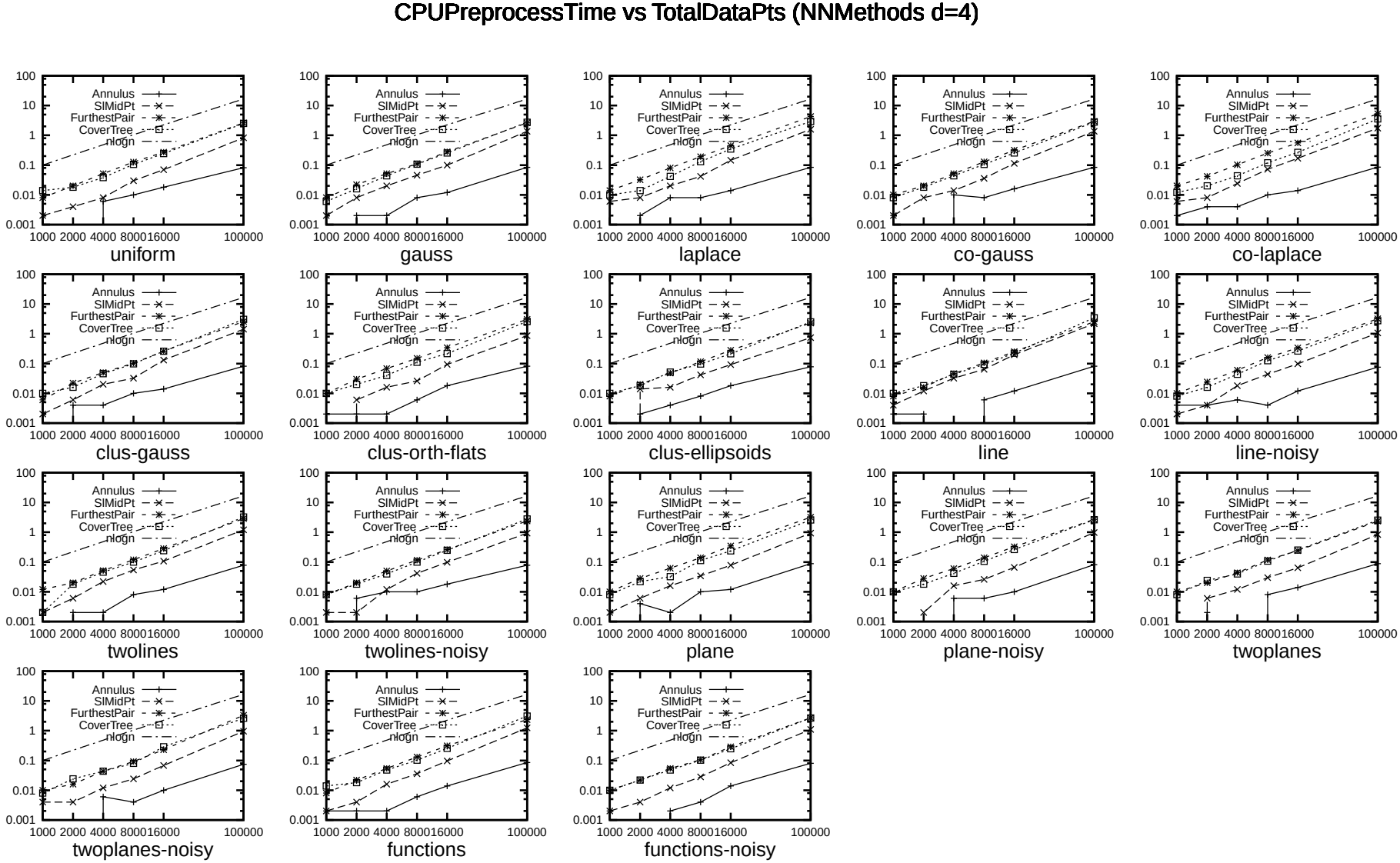


Figure 4.14: Preprocessing time of NN methods for increasing n.

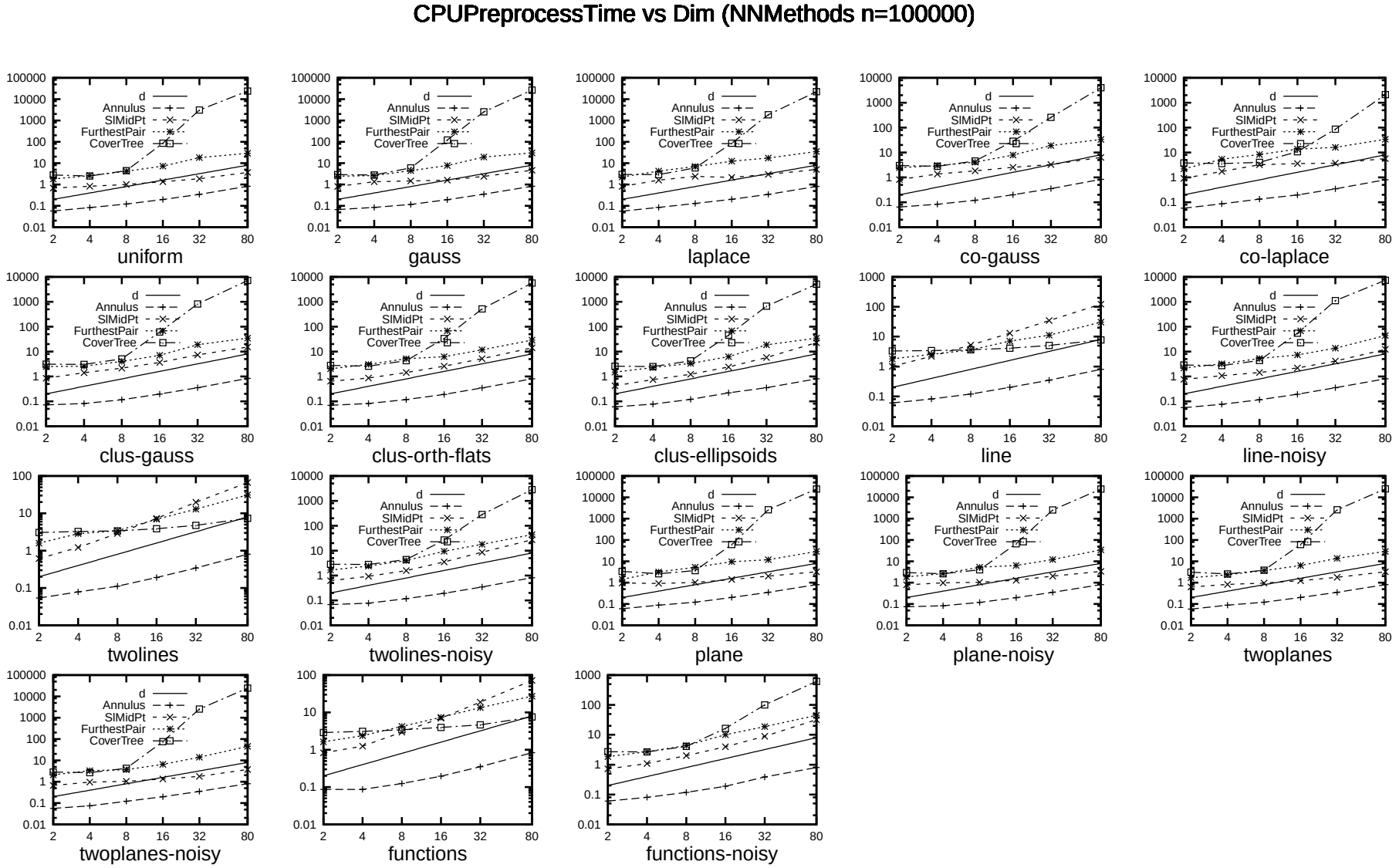


Figure 4.15: Preprocessing time of NN methods for increasing d.

$n = 100000$.

Figures 4.18[3] and 4.19[4], for respectively non-uniform and uniform query, plot the average data points visited by the methods for increasing d, for $k = 5$ and $n = 100000$. It can be seen for non-uniform query in Figure 4.18[3] that for distributions other than non-noisy lines, KDTree is the best method for lower d's (< 16), whereas at higher d's Cover Tree is either the best or one of the best methods. Furthermore, Metric Tree on these distributions performs better than Cover Tree for lower d's but becomes worse as d gets higher. This behaviour, however, is only on the selected high value of n of the figure, as Cover Tree performs worse than Metric Tree (and KDTree) for $n = 100000$, as was seen in Figure 4.16[1]. On non-noisy line distributions, it can be seen, that Annulus Method and Metric Tree are better than others. It can be seen that Annulus Method is the worst over all, it is only better on non-noisy lines and for higher d's on noisy straight lines, while Metric Tree is worse than KDTree for lower d's, and worse or similar to Cover Tree for higher d's on all but the non-noisy line distributions. For uniform query, it can be seen in Figure 4.19[4], Annulus Method is the worst overall; in many cases even at moderate d values it completely degrades to Linear Search. Overall Cover Tree is the best method. On distributions other than non-noisy lines, Metric Tree is mostly similar to Cover Tree at low and high d's, but mostly worse for middle d's. KDTree also shows the similar behaviour but in many cases is even worse than Metric Tree, and also sometimes worse than Cover Tree and Metric Tree at high d's. On non-noisy line distributions, Metric Tree has the same rate of growth as Cover Tree but is a constant times worse, whereas KDTree has a much higher growth rate and becomes worse than both the two as d increases.

Overall in summary, from figures 4.16[1], 4.17[2], 4.18[3], and 4.19[4], for non-uniform query KDTree is the best for lower d's (< 16) and Cover Tree is better for higher d's. For uniform query Cover Tree is the best. Annulus Method is the worst method overall; only for non-uniform query on non-noisy straight lines it is good. Metric Tree offers performance very similar to Cover Tree but is worse at higher d's. Cover Tree, however, has the drawback that it degrades faster than the others for increasing n on non-uniform query. All methods can not cope with increasing d's and degrade (completely) to Linear Search on all but the clustered and line distributions, for both non-uniform and uniform query.

Let us now look at how well the query performance of the methods measured in average points visited matches the CPU query time. Figures 4.20[5], 4.21[6], 4.22[7] and 4.23[8], are respectively equivalent to 4.16[1], 4.17[2], 4.18[3] and 4.19[4], but are plotted for CPU query time rather than average points visited. It can be seen from the figures that the

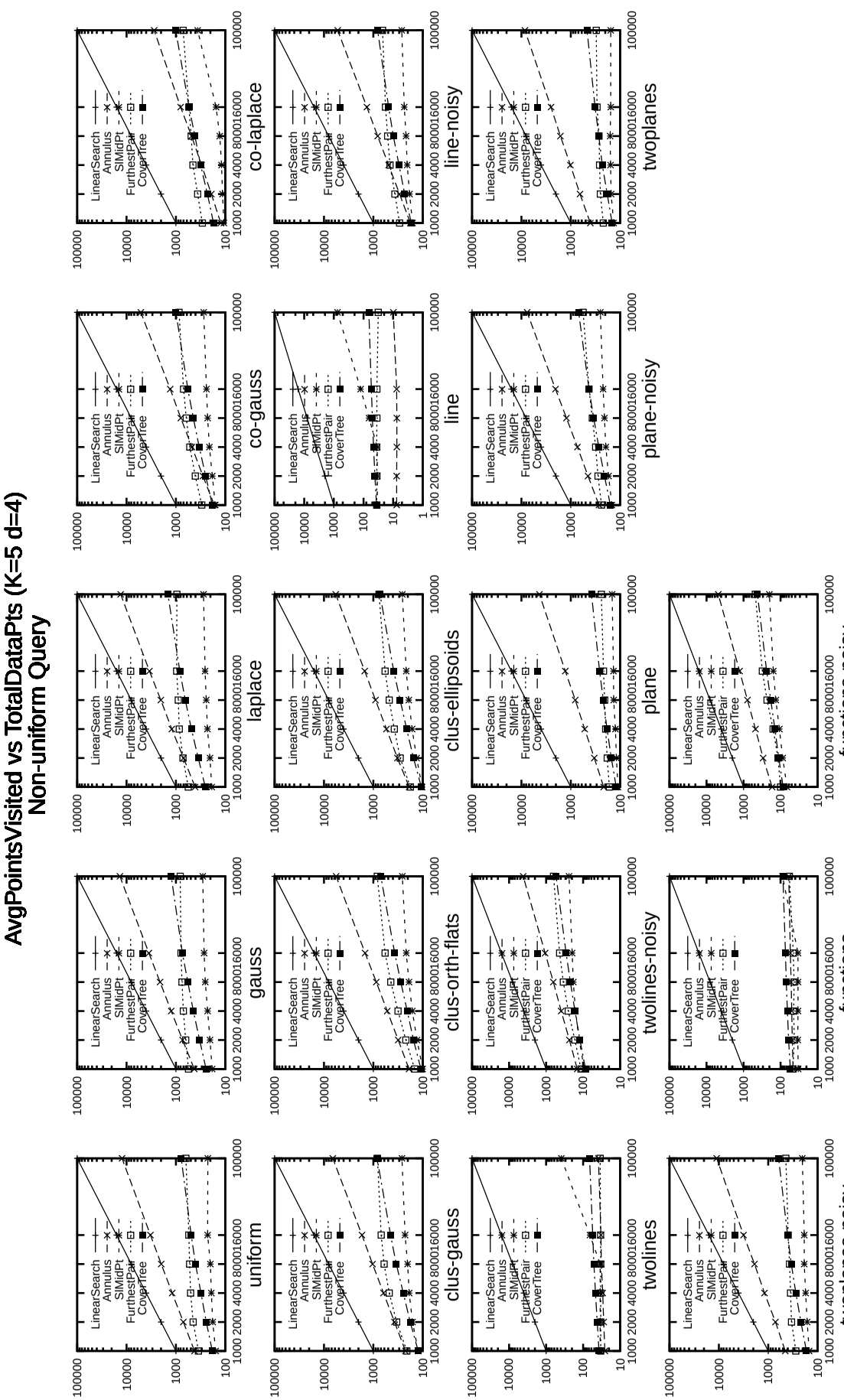


Figure 4.16: Avg points visited by NN Methods for increasing n on non-uniform query.

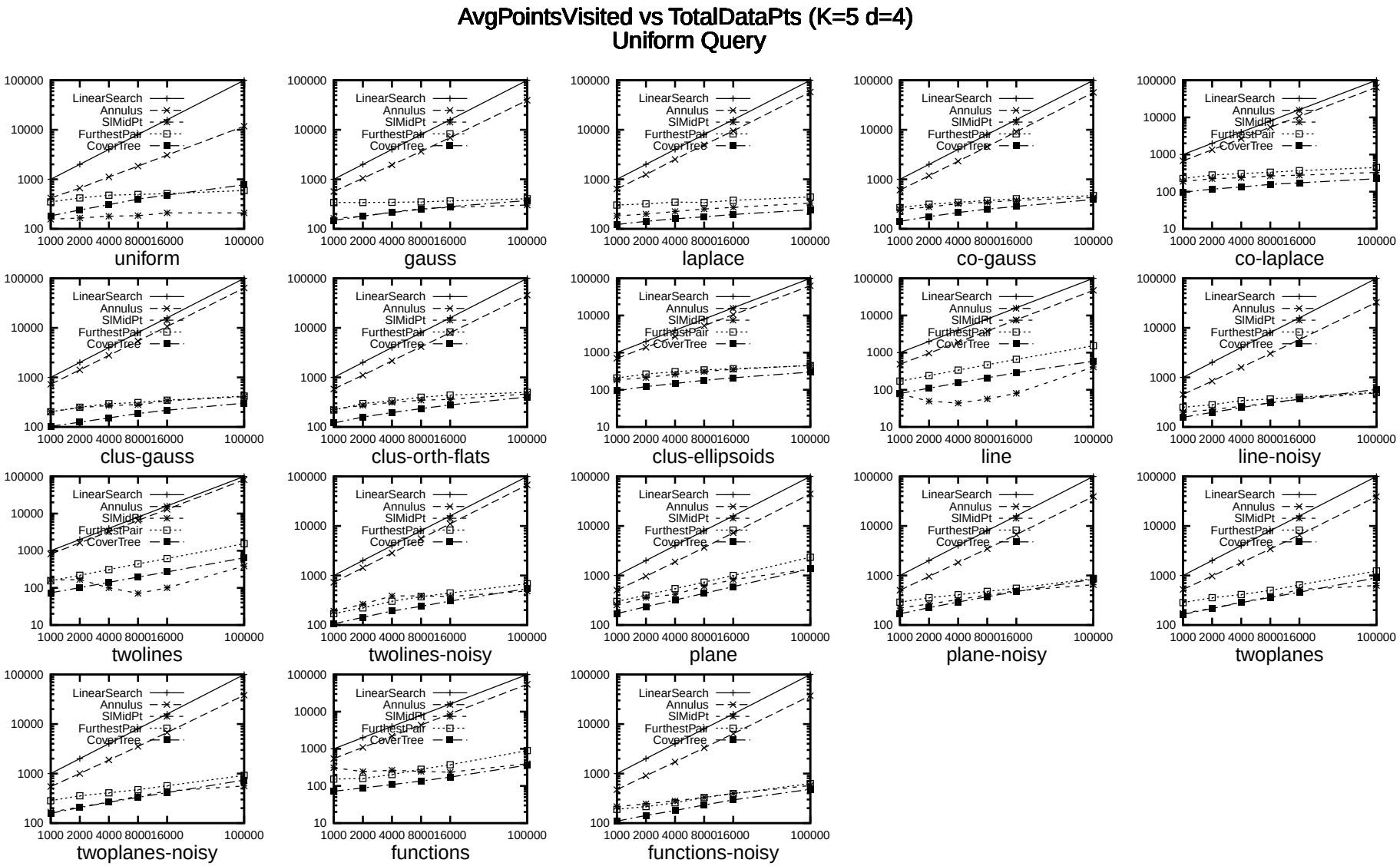


Figure 4.17: Avg points visited by NN Methods for increasing n on uniform query.

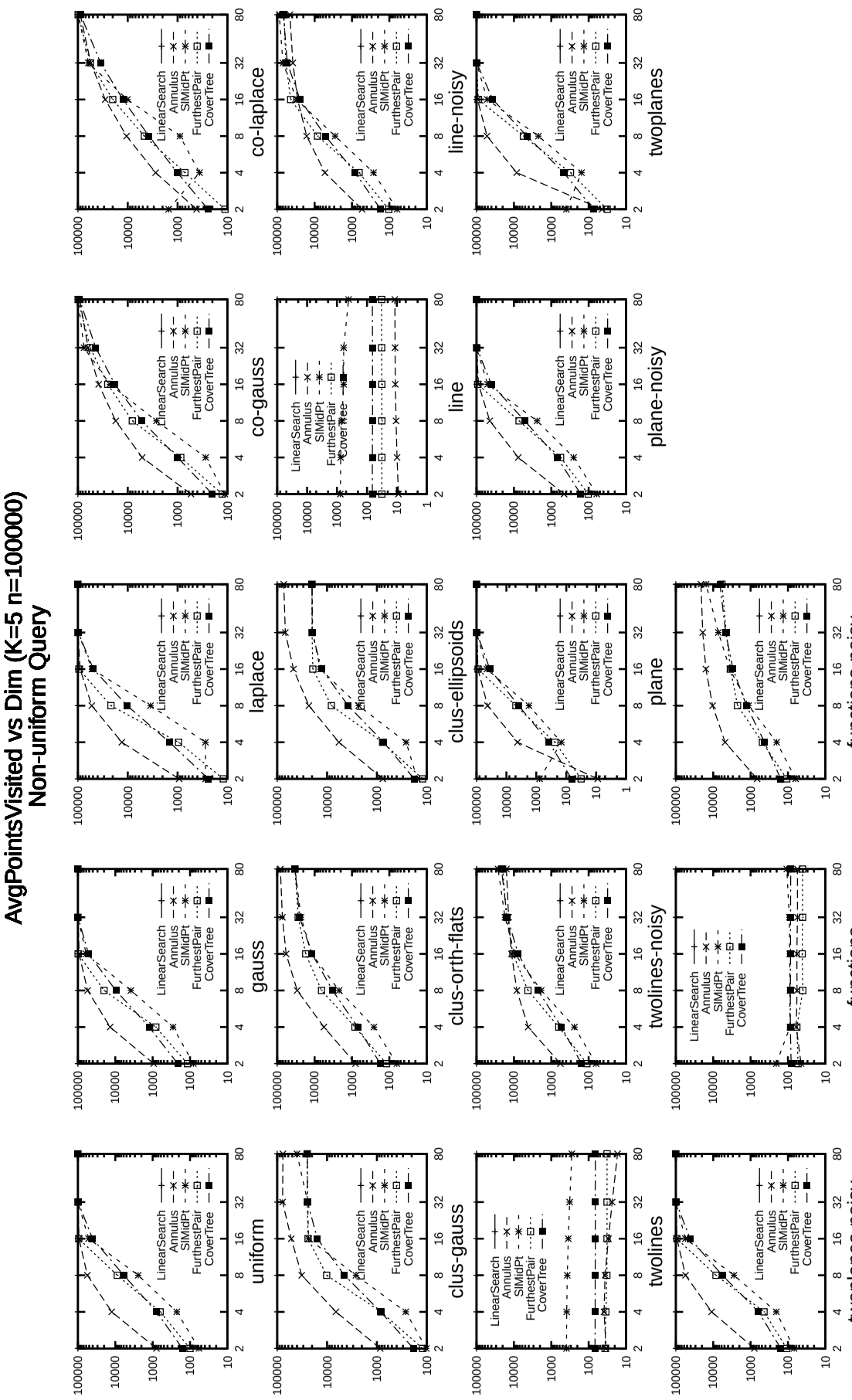


Figure 4.18: Avg points visited by NN Methods for increasing d on non-uniform query.

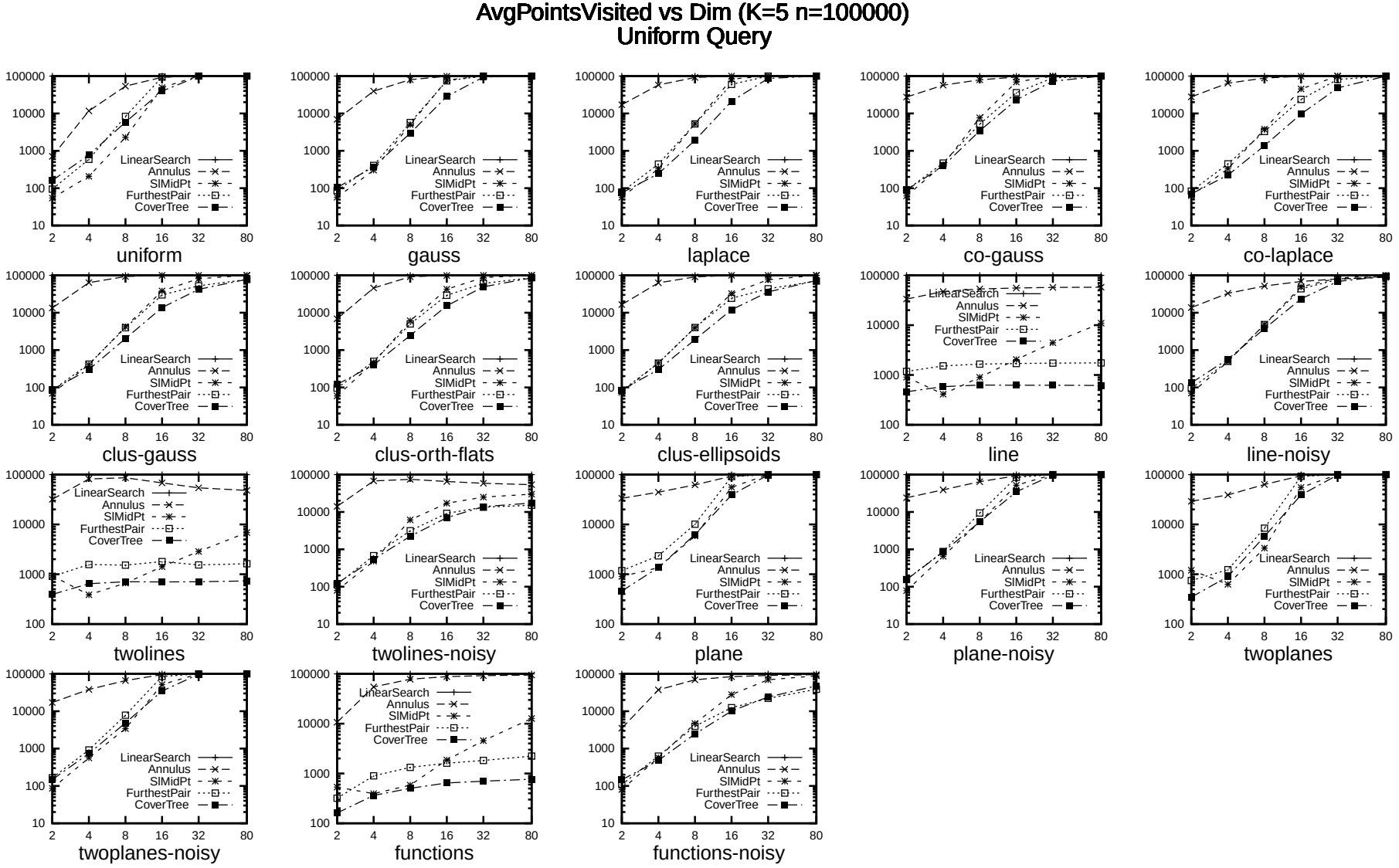


Figure 4.19: Avg points visited by NN Methods for increasing d on uniform query.

extra overhead involved in pruning away points from inspection starts to show up, and the methods get much closer to Linear Search in CPU query time than they were in average points visited. Cover Tree especially, because of it larger tree size relatively gets higher than the other trees (though strangely its growth rate become lower for increasing n). In Figure 4.21[6], for increasing n on uniform query, it is no longer the best method for $n < 100000$ like in Figure 4.17[2], infact it is the worst method compared to KDTree and Metric Tree, and KDTree, probably because of its simpler and faster pruning procedure, is the best method. Also, it can be seen in Figure 4.21[6], that on most distributions Annulus Method gets even worse than Linear Search for higher n's, while it was not the case in Figure 4.17[2]. With respect to d it can be seen in Figure 4.22[7], that for non-uniform query KDTree (on distributions other than non-noisy lines) is not only still the best for lower d's but becomes better or equivalent to others even at moderate and higher values of d. For uniform query (Figure 4.23[8]) Cover Tree is still the best for moderate d values, but not any more on planar distributions, where KDTree now becomes the best even at high d's. All the methods, however, are of not much use for $d > 16$ on distributions other than the clustered and non-noisy lines, as for both non-uniform and uniform query they become even worse than the Linear Search for $d > 16$. Annulus Method, though, gets worse even sooner, sometimes even at $d = 2$; thus performing much poorer than what was initially hoped. Trends similar to what has been described were also observed in plots with other fixed values of k, d, and n.

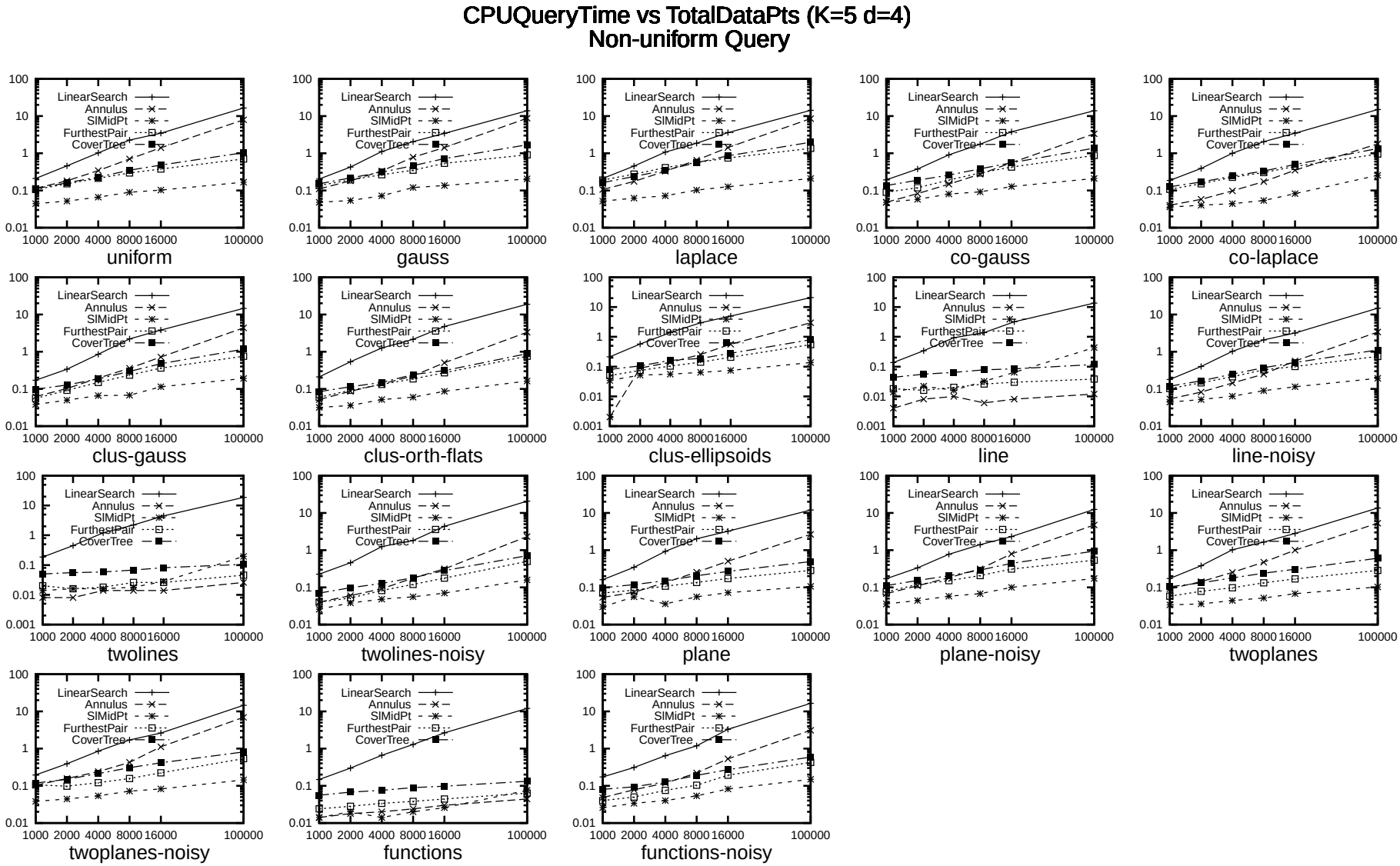


Figure 4.20: CPU query time of NN Methods for increasing n on non-uniform query.

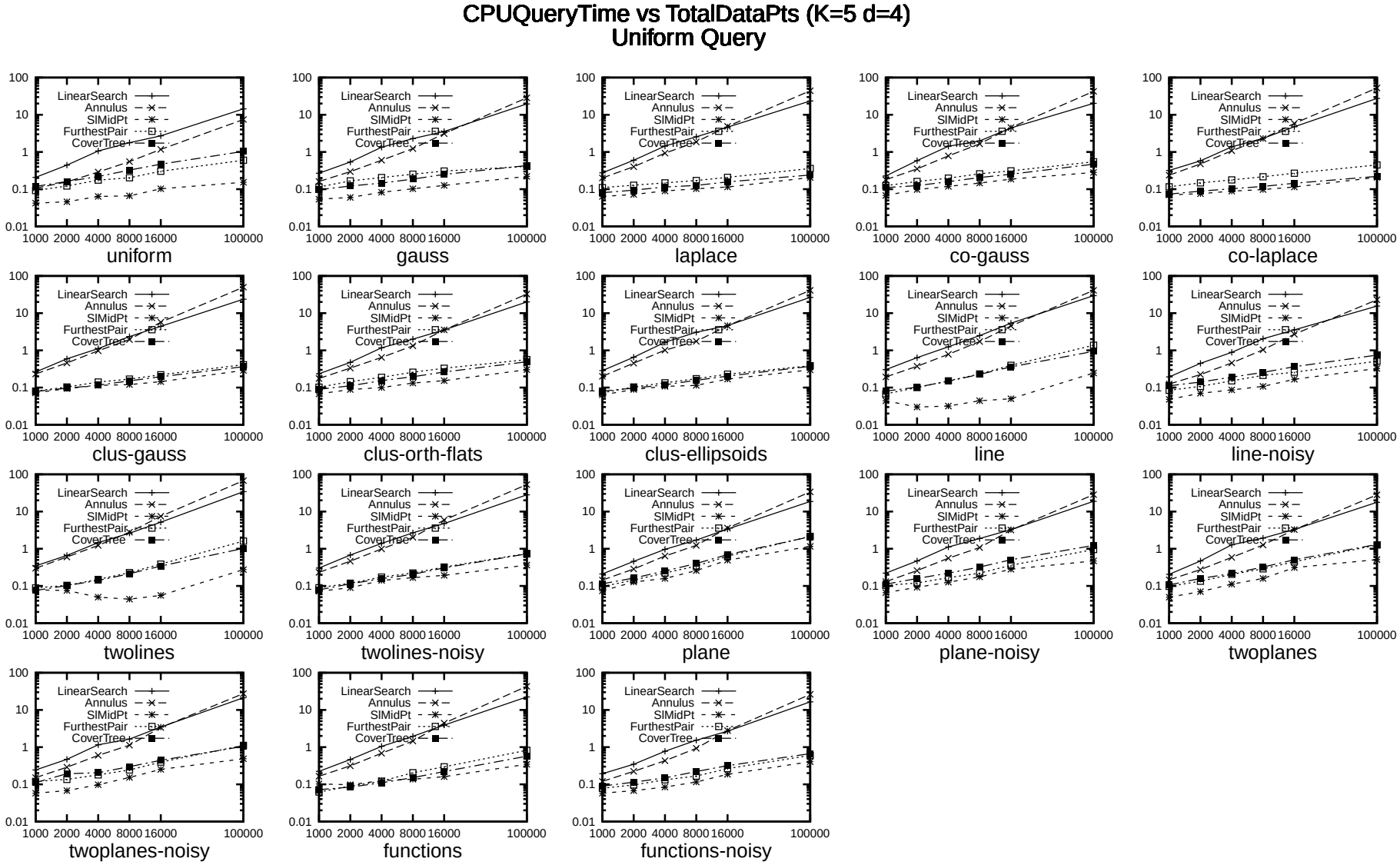


Figure 4.21: CPU query time of NN Methods for increasing n on uniform query.

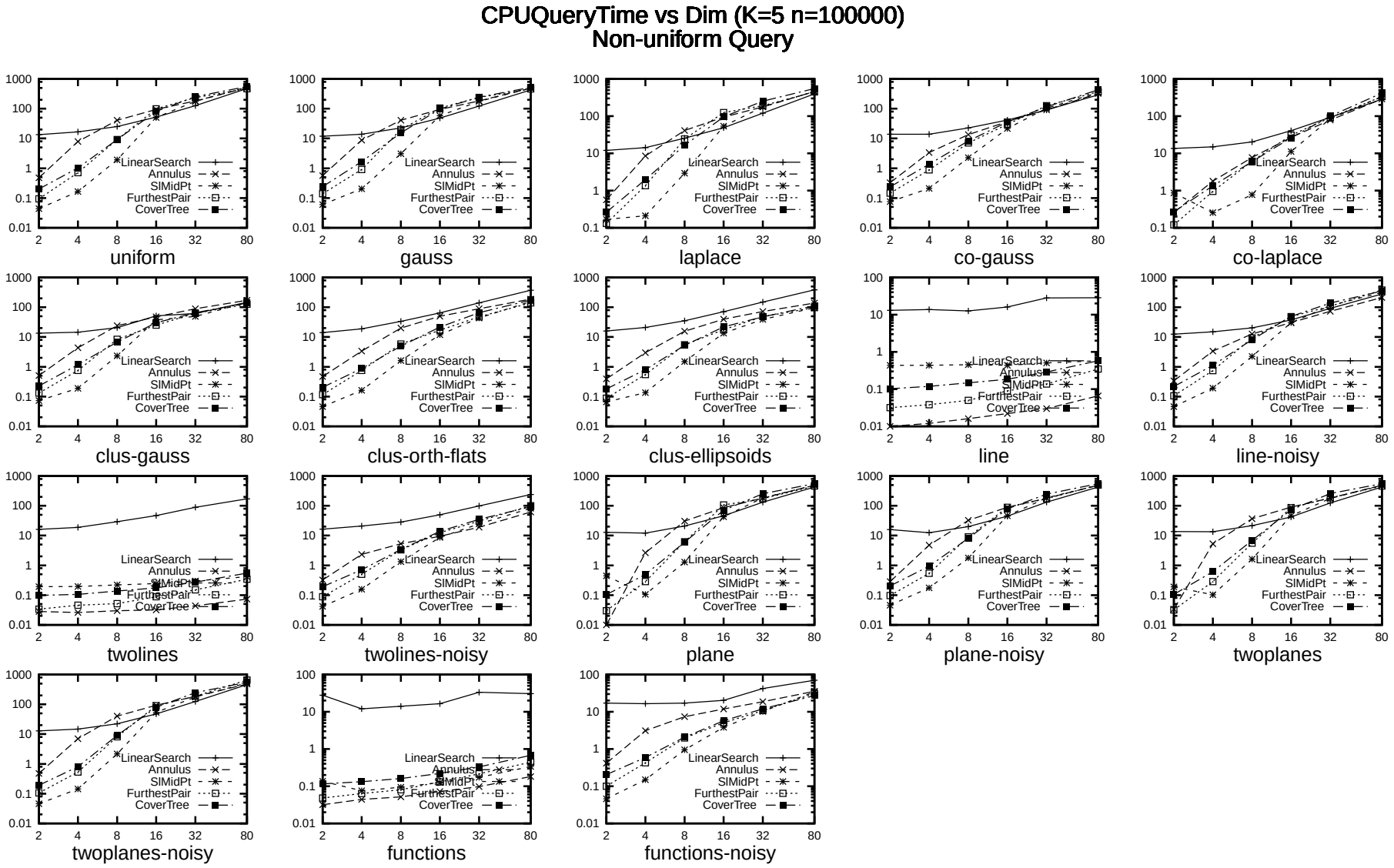


Figure 4.22: CPU query time of NN Methods for increasing d on non-uniform query.

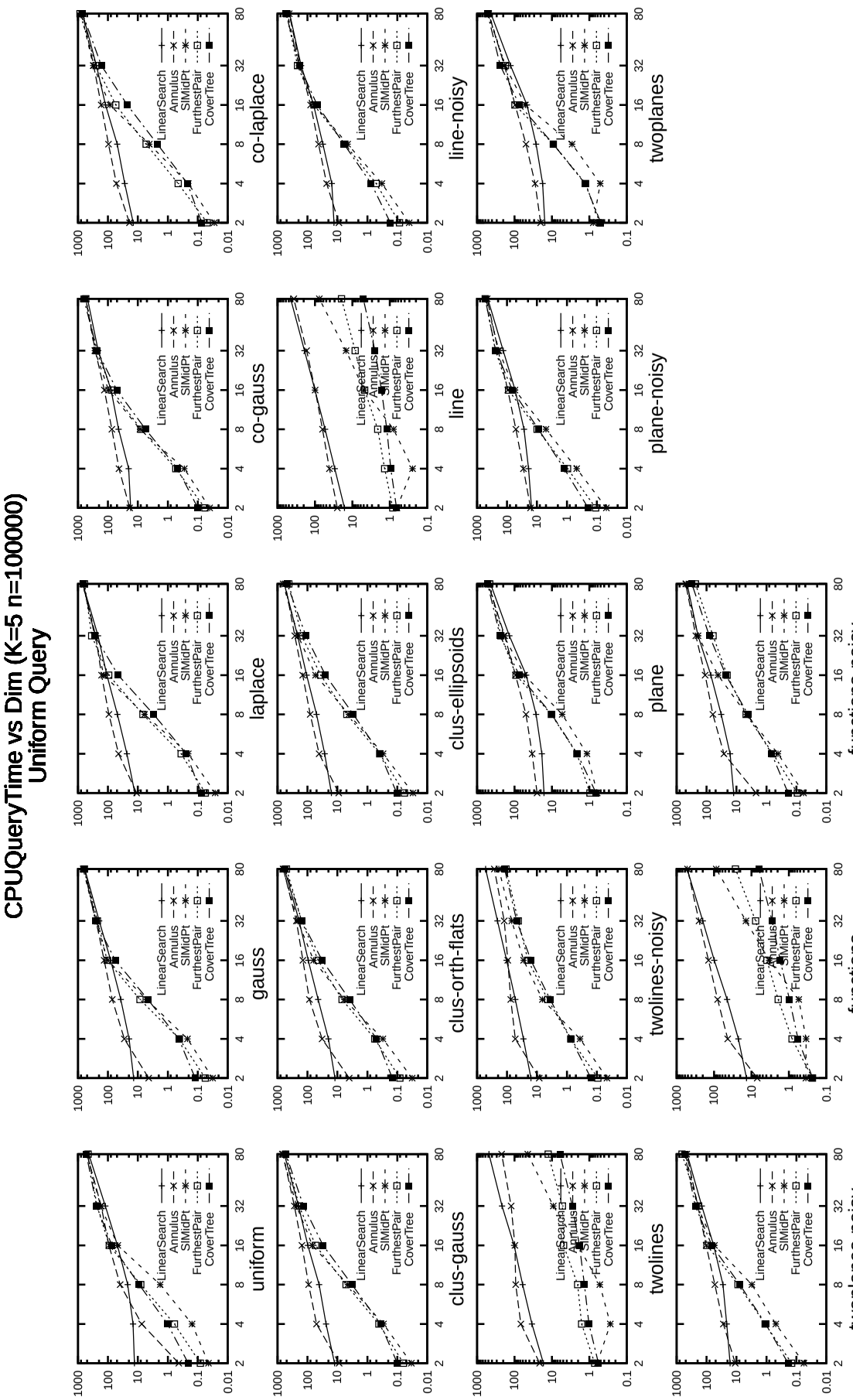


Figure 4.23: CPU query time of NN Methods for increasing d on uniform query.

Chapter 5

Conclusion

Since the initial inception of the NN problem a large number of techniques have been proposed for its solution. However, ideal solutions exist only for $d \leq 2$, and solutions for $d > 2$ are still far from ideal. All the proposed solutions for $d > 2$, that have reasonable space and preprocessing time, degrade to Linear Search for d's.

For moderate d's (≤ 10), KDTrees are one of the oldest and most popular techniques proposed for NN search. Metric Trees are a popular and newer technique, which are claimed to be the state-of-the-art for moderate d's. More recently, Cover Trees have been proposed, which are designed to exploit the low intrinsic dimensionality of points embedded in higher dimensions, and are thus claimed to give better performance even at higher d's. All these techniques, however, have not been thoroughly compared with each other, and their performance relative to each other was essentially unknown. Moreover, the two most popular techniques, KDTrees and Metric Trees, have a number proposed construction methods, which have not been compared with each other either. This thesis dealt with a thorough empirical investigation of the various construction methods for KDTrees and Metric Trees, and also, comparison of the two trees against each other, and against Cover Trees and the Annulus Method.

It was observed in the investigation carried out for this thesis that all the evaluated techniques suffer from the curse-of-dimensionality and generally become worse than Linear Search for $d > 16$. Excluding the Annulus Method, which always performed poorly, the rest of methods are only better than Linear Search if the points are clustered or lie on a line. KDTree is the best method if the query points have the same distribution as that of the data, otherwise KDTree is best for low d's but for higher d's Cover Tree is the best. KDTree is worse than the other methods only for points lying on a line, a case which is uncommon in practice. In other cases, even if we look at points visited rather than raw CPU times, KDTree is either better or very similar to Metric Tree. Hence, the state-of-the-art claim of Metric Trees could not be verified. Cover Trees, though more sophisticated and claimed to be better at higher d's, performed very similar to Metric Trees in this

investigation. Only at moderate to high d's, they are somewhat better than Metric Trees. Cover Trees also appear to have a high query time overhead, as they are slightly better off in terms of points visited than CPU query time, compared to Metric Trees and the others. The Annulus Method, which is an old technique but was rediscovered during the research performed for this thesis, was found to be the worst overall; only in the pathological case of points lying on a line it gives better query performance than the others.

Among the construction methods for KDTrees, Sliding Midpoint of Widest Side (SlMidPt) was found to be the best. It gives better query performance when query points have a different distribution than the data, and also works equally well if their distribution is the same. Among the construction methods for Metric Trees, Points Closest to Further Pair (FurthestPair) was found to give optimal or near optimal query performance, with little additional overhead in preprocessing/construction time.

Appendix A

Additional Results for KDTrees

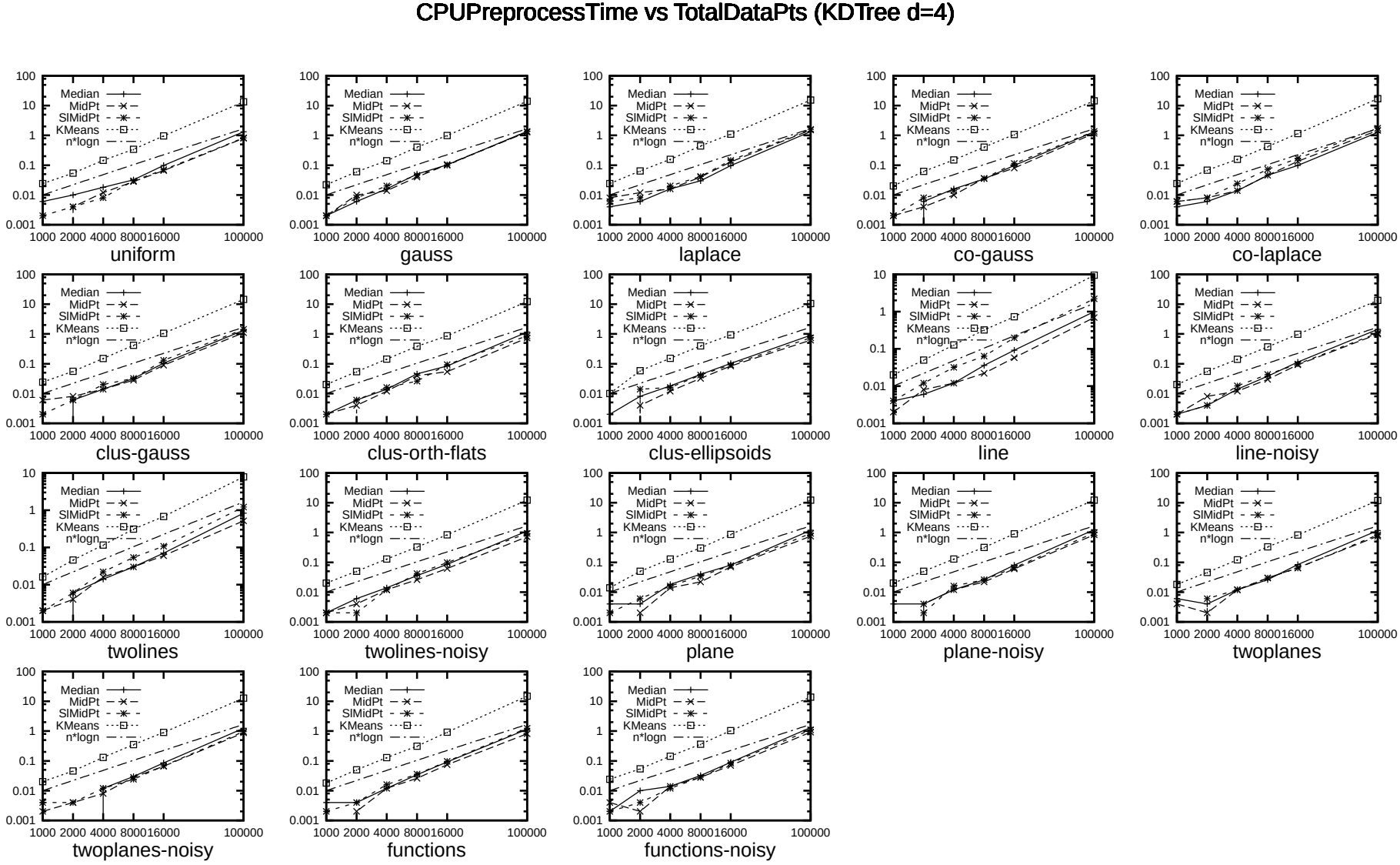


Figure A.1: KDTrees' construction time for increasing n.

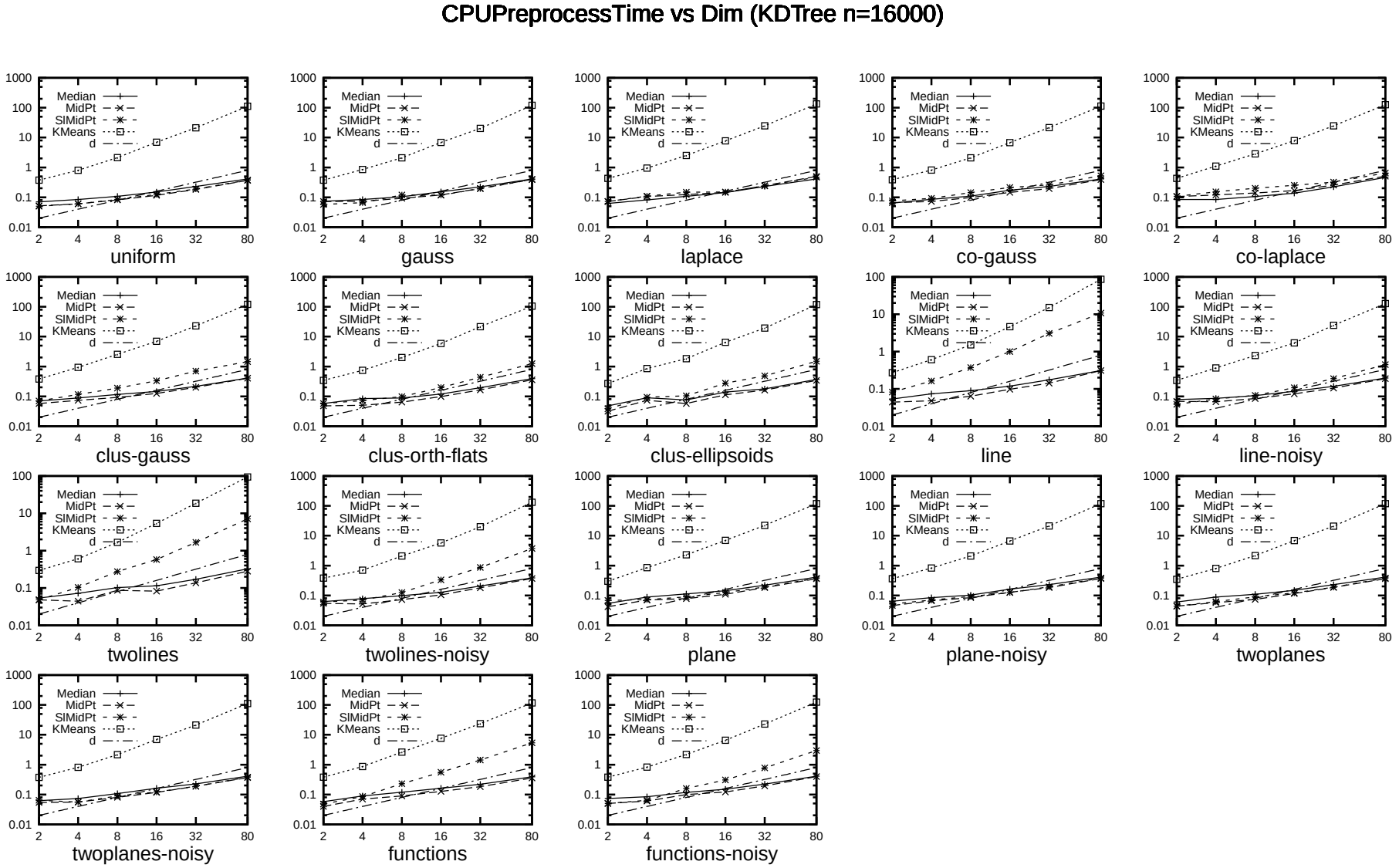


Figure A.2: KDTrees' construction time for increasing d.

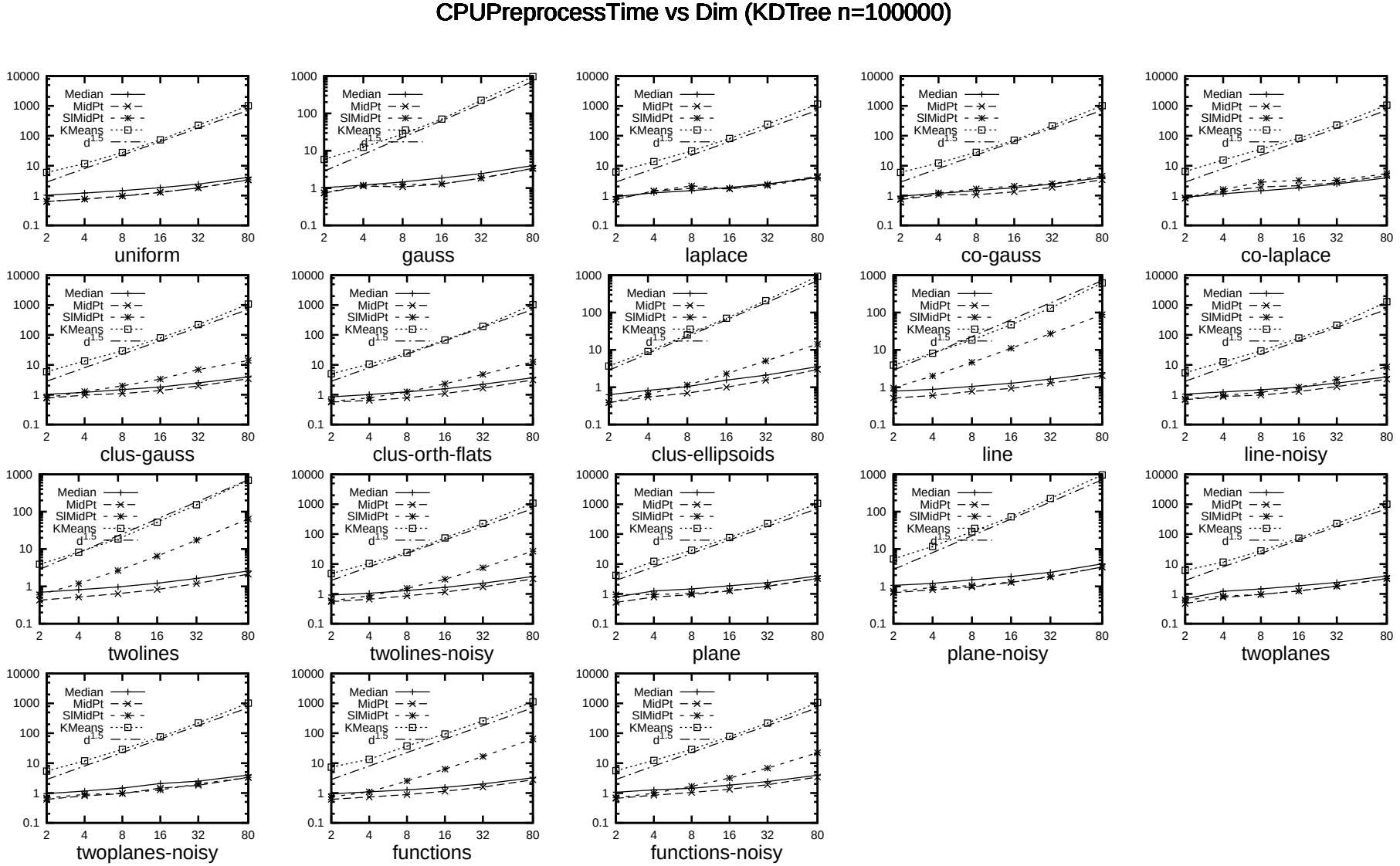


Figure A.3: KMeans $O(d^{1.5})$ construction time.

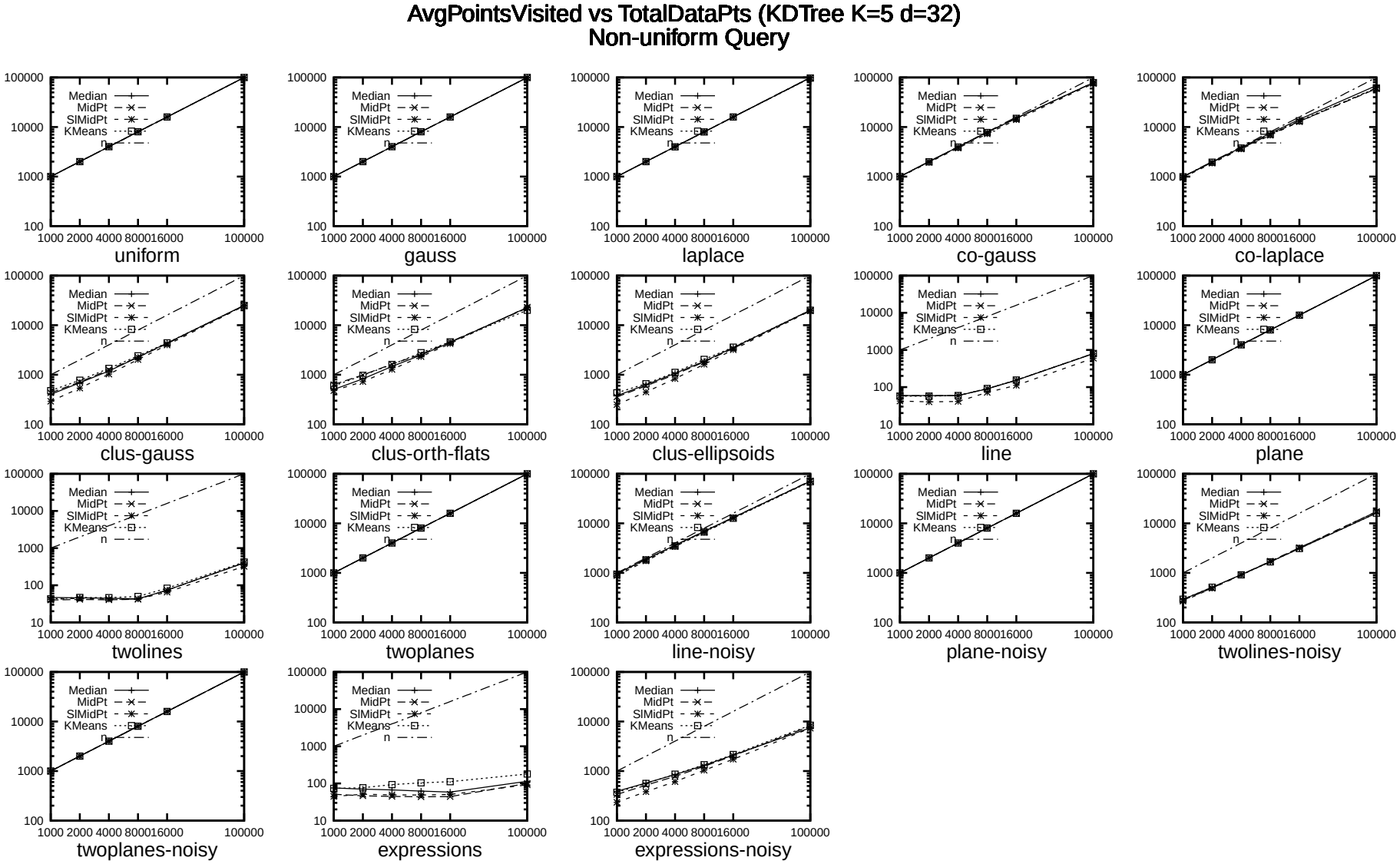


Figure A.4: Degradation of KDTrees towards n at higher d's.

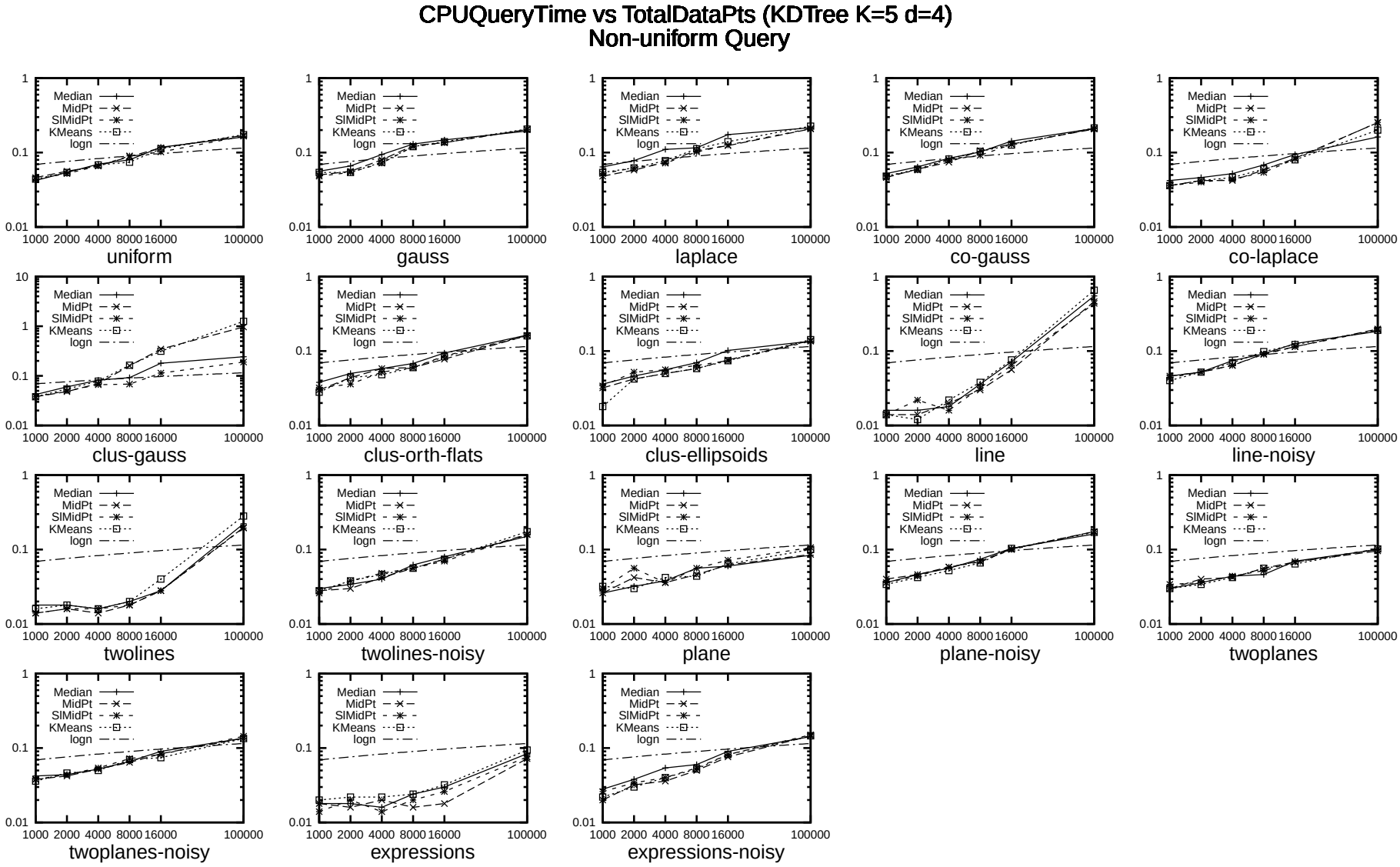


Figure A.5: CPU query time of KDTrees for increasing n on non-uniform query.

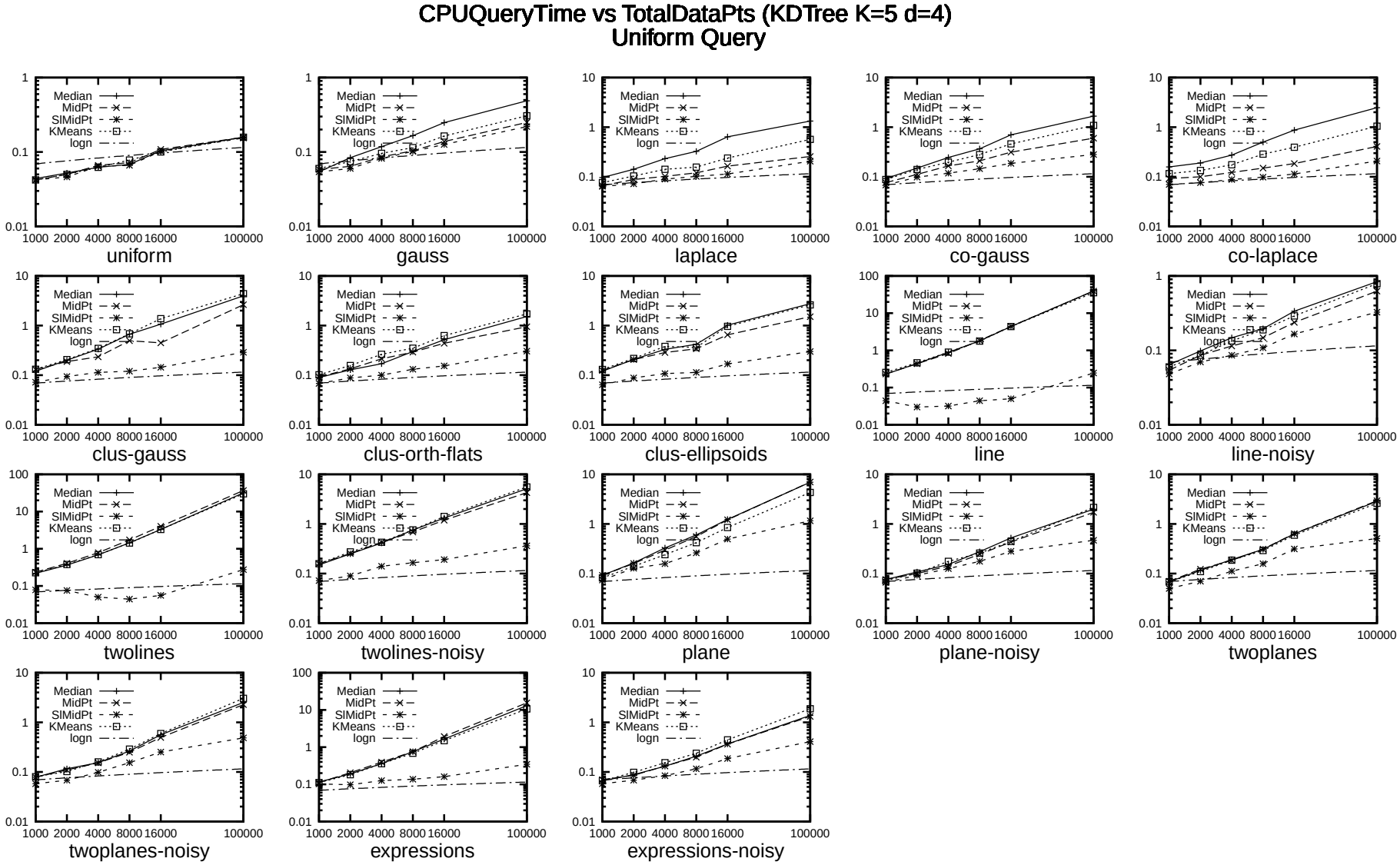


Figure A.6: CPU query time of KDTrees for increasing n on uniform query.

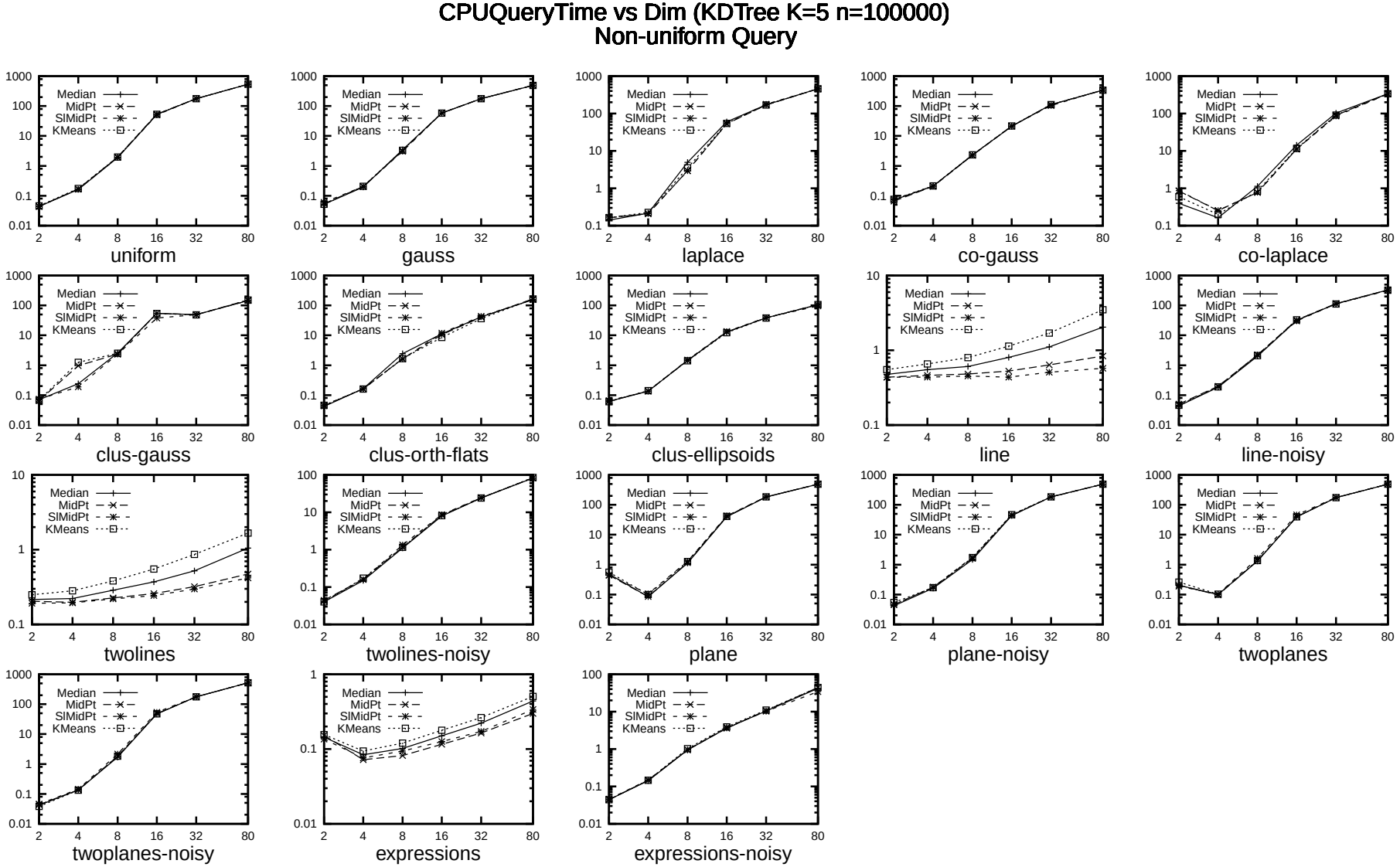


Figure A.7: CPU query time of KDTrees for increasing d on non-uniform query.

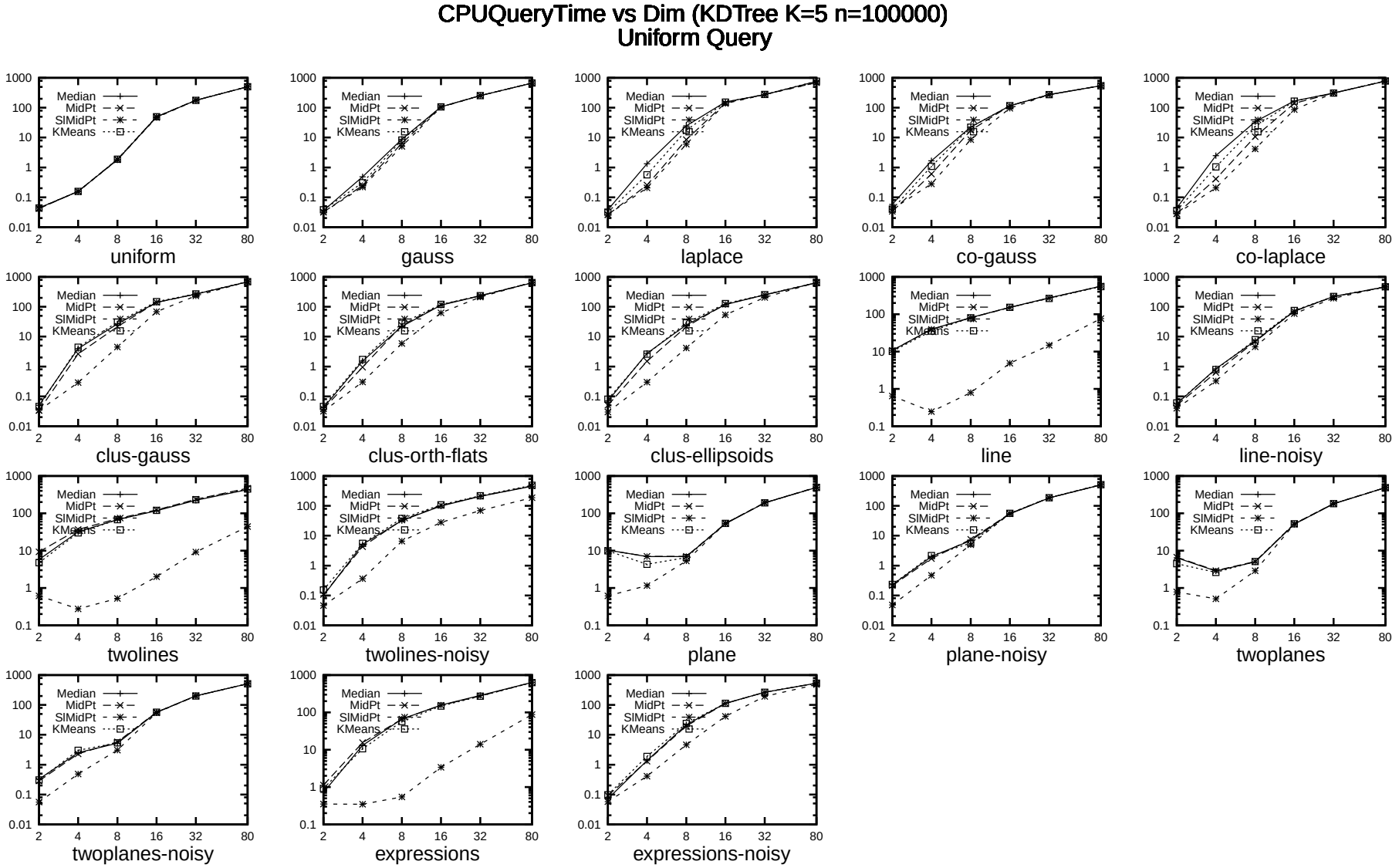


Figure A.8: CPU query time of KDTrees for increasing d on uniform query.

Appendix B

Additional Results for Metric Trees

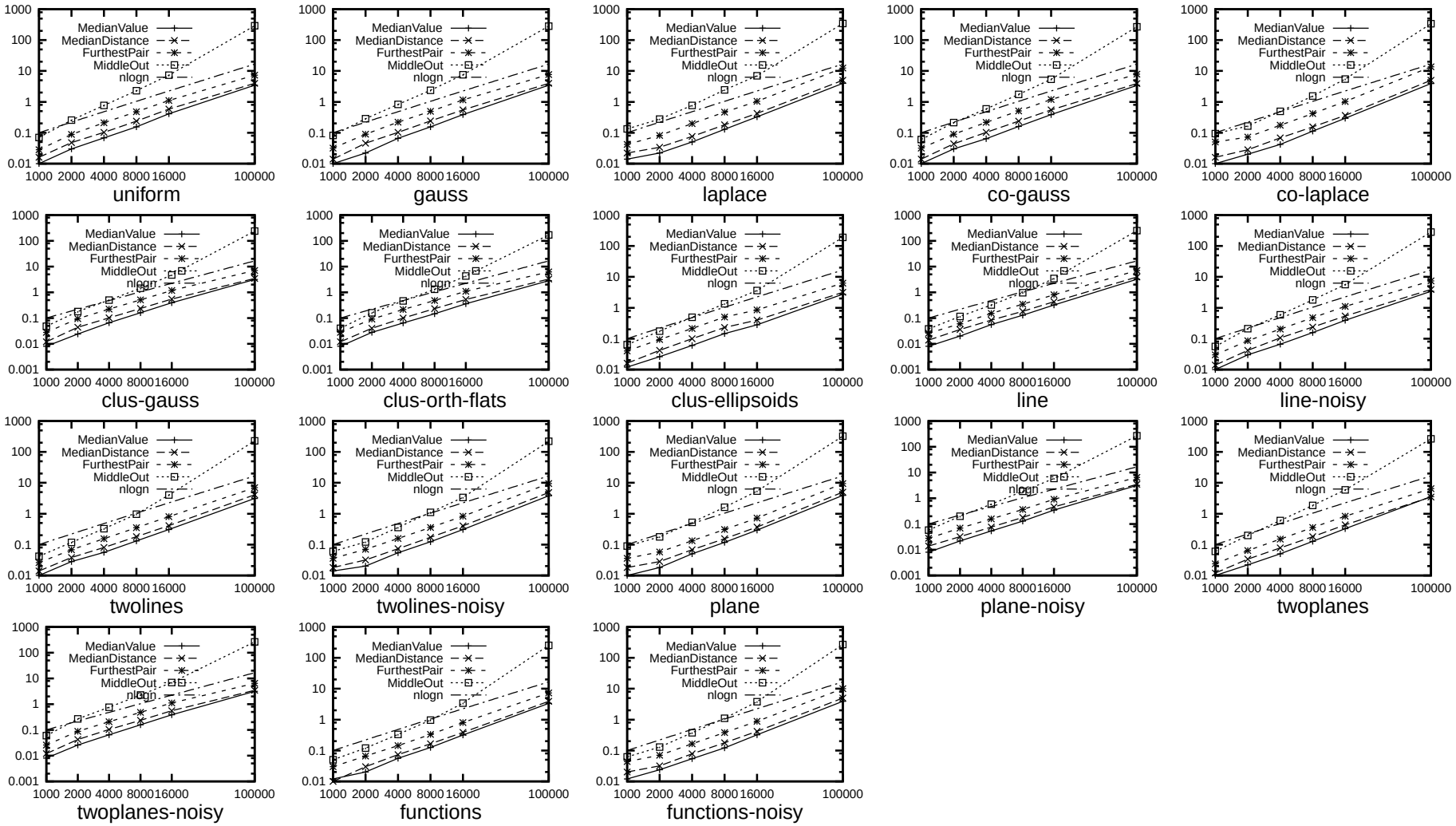


Figure B.1: Metric Trees' construction time for increasing n with d=16.

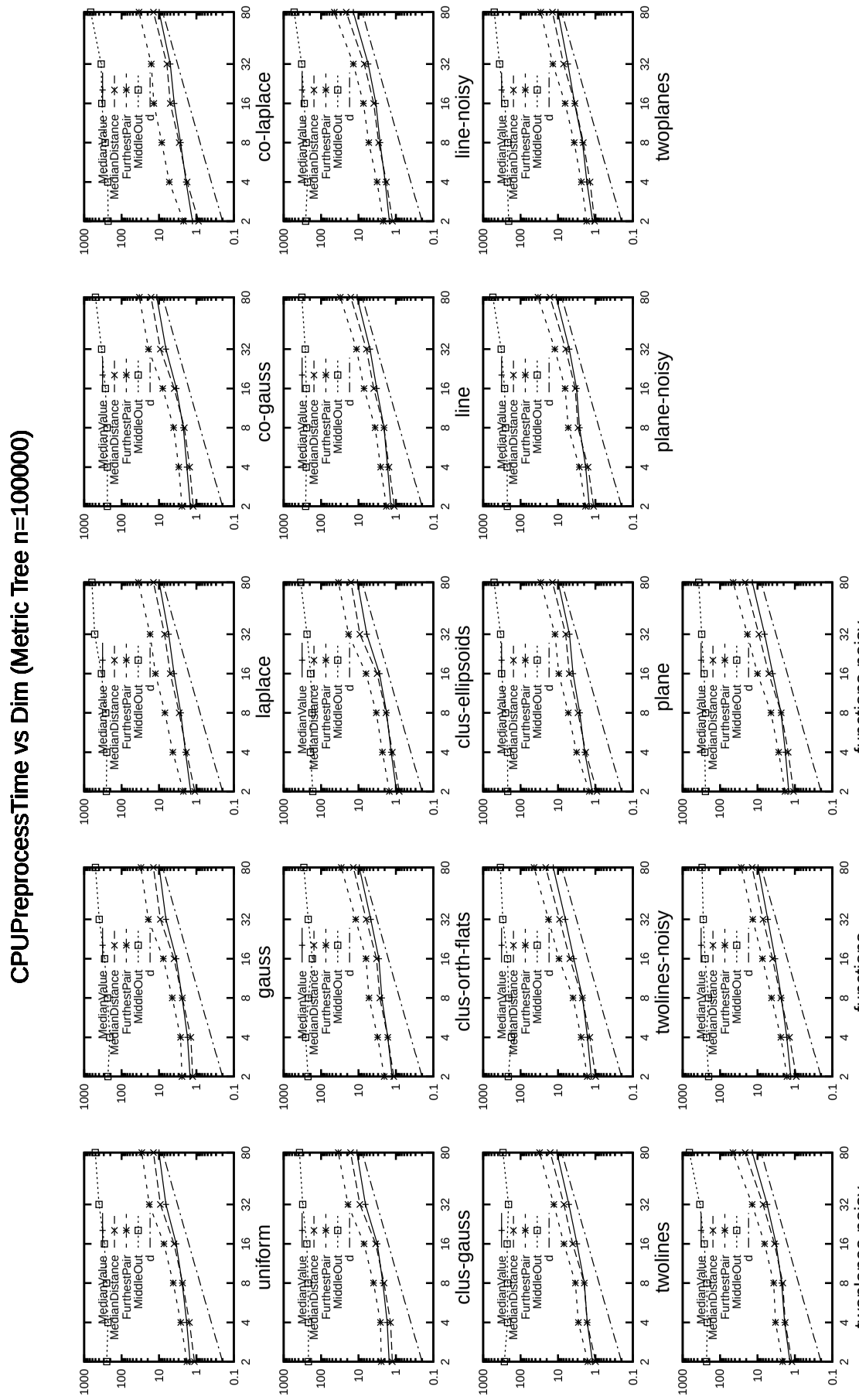


Figure B.2: Metric Trees' construction time for increasing d with n=100K.

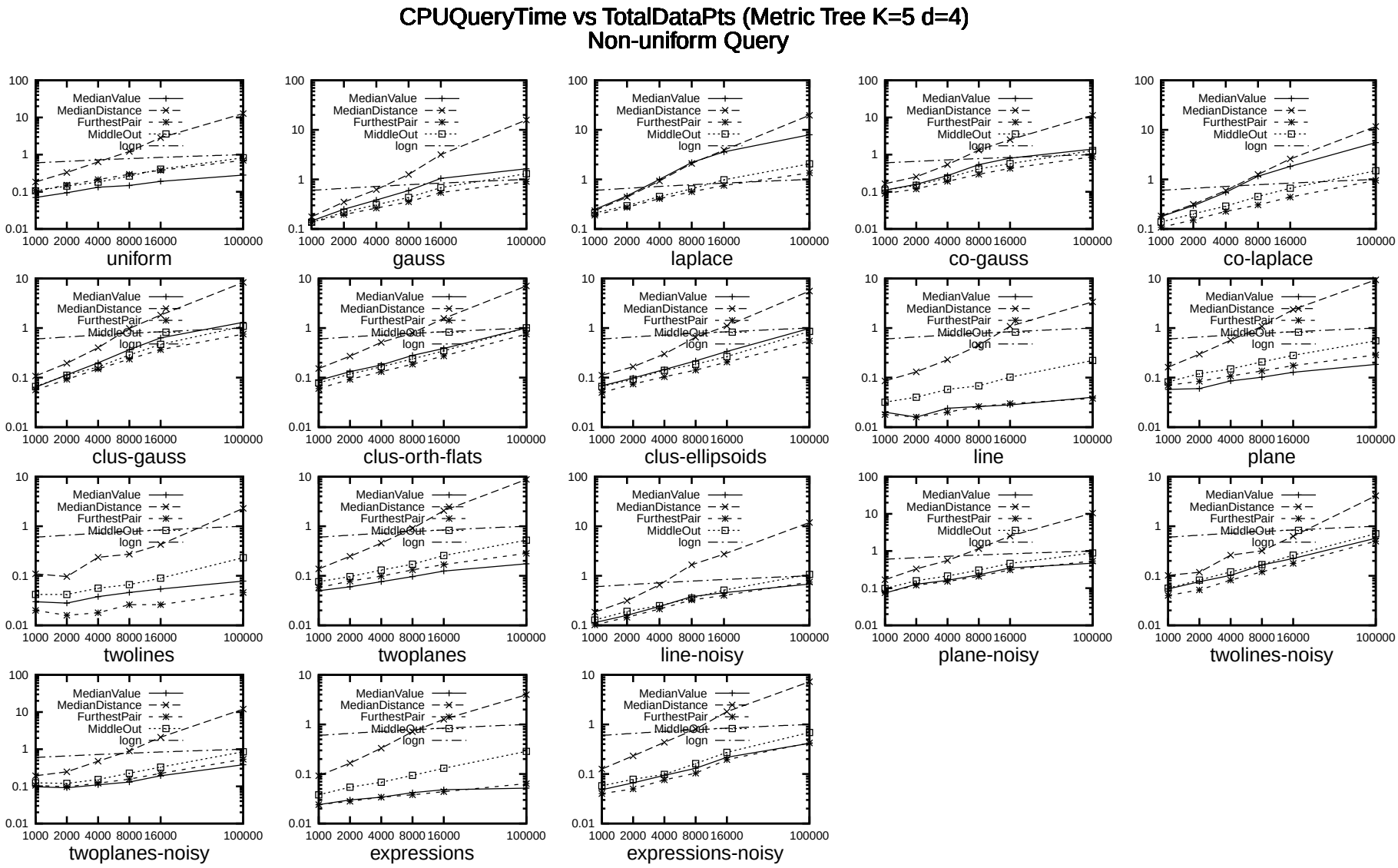


Figure B.3: CPU query time of Metric Trees for increasing n on non-uniform query.

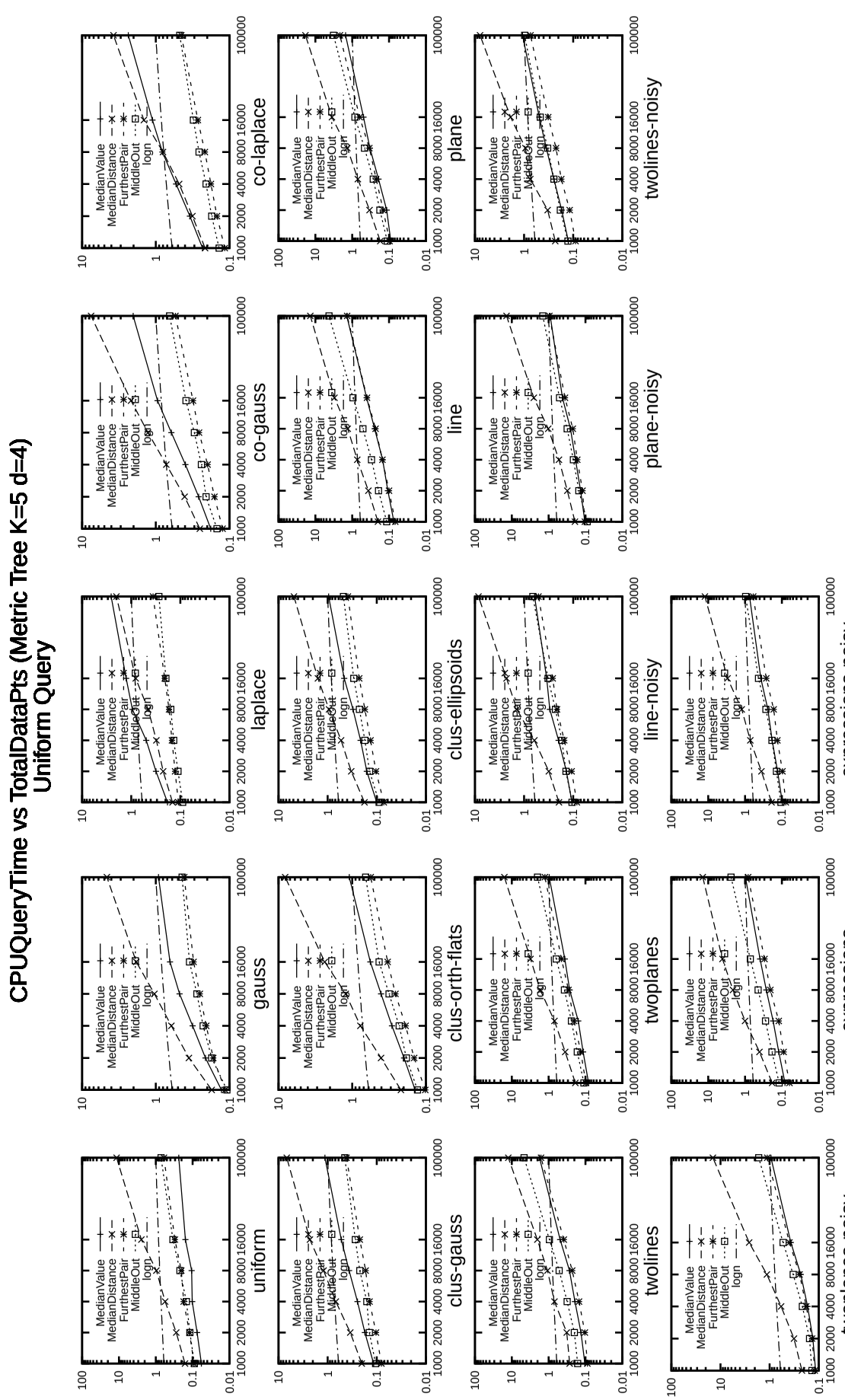


Figure B.4: CPU query time of Metric Trees for increasing n on uniform query.

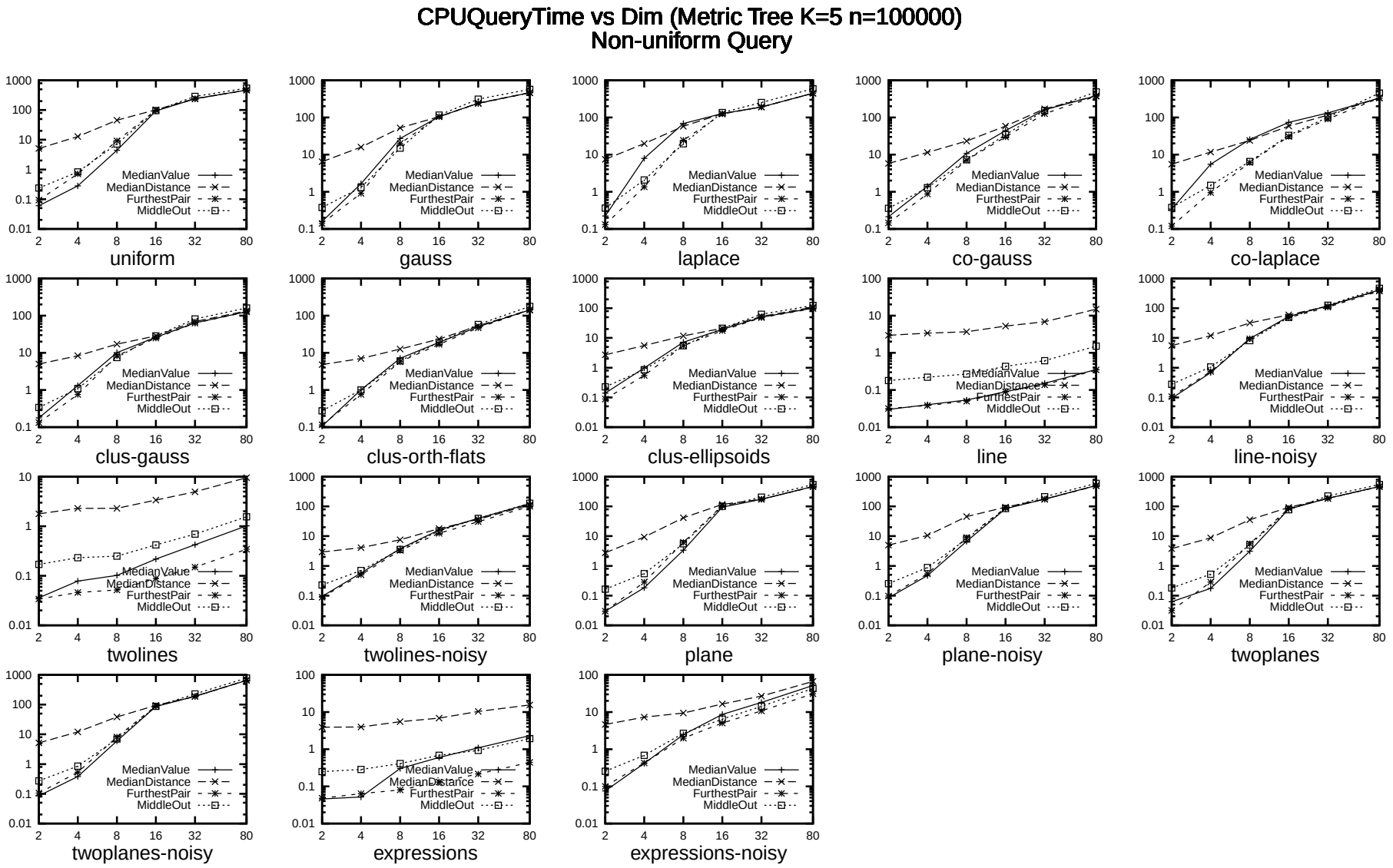


Figure B.5: CPU query time of Metric Trees for increasing d on non-uniform query.

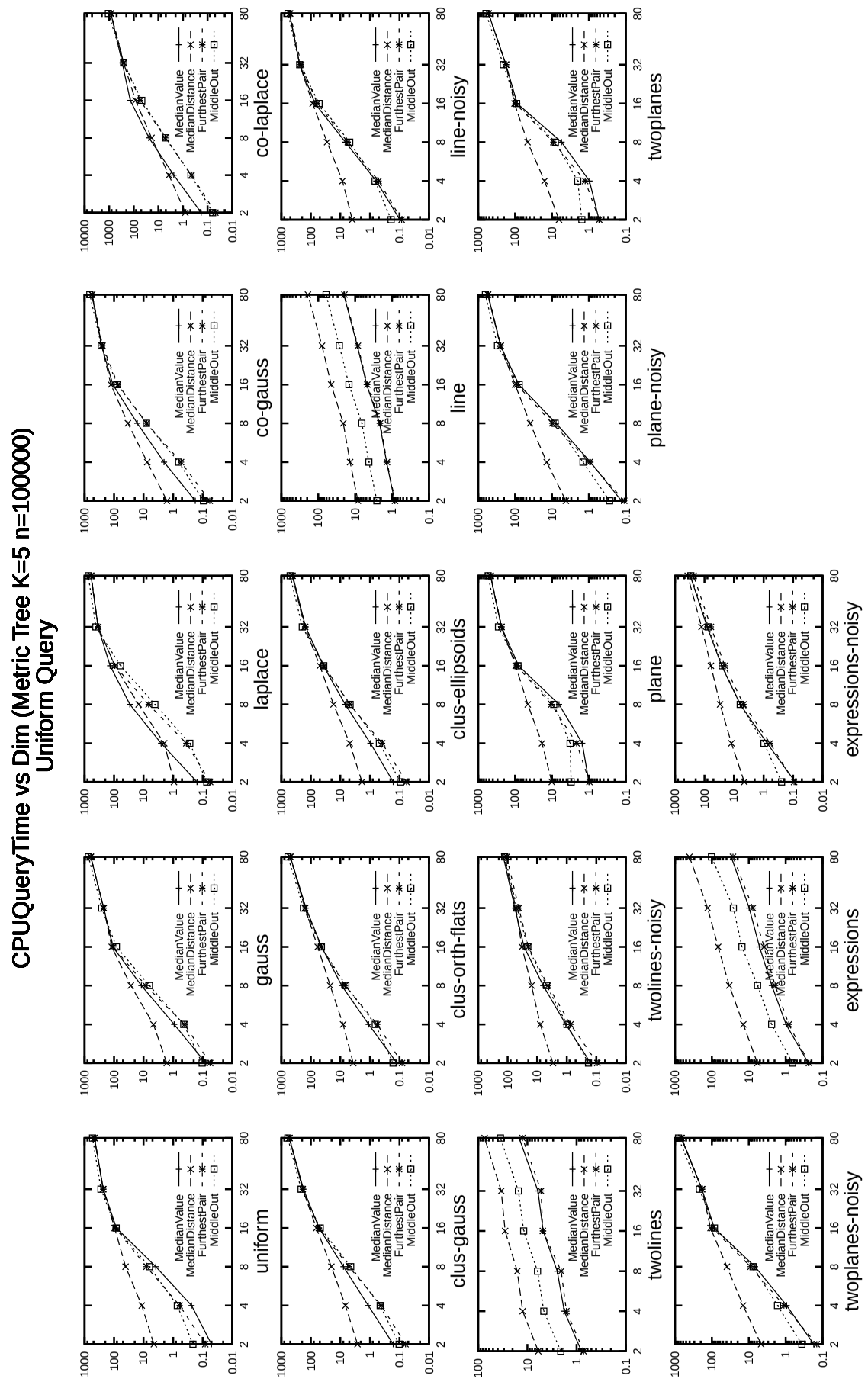


Figure B.6: CPU query time of Metric Trees for increasing d on uniform query.

Appendix C
Additional Results for NN Methods

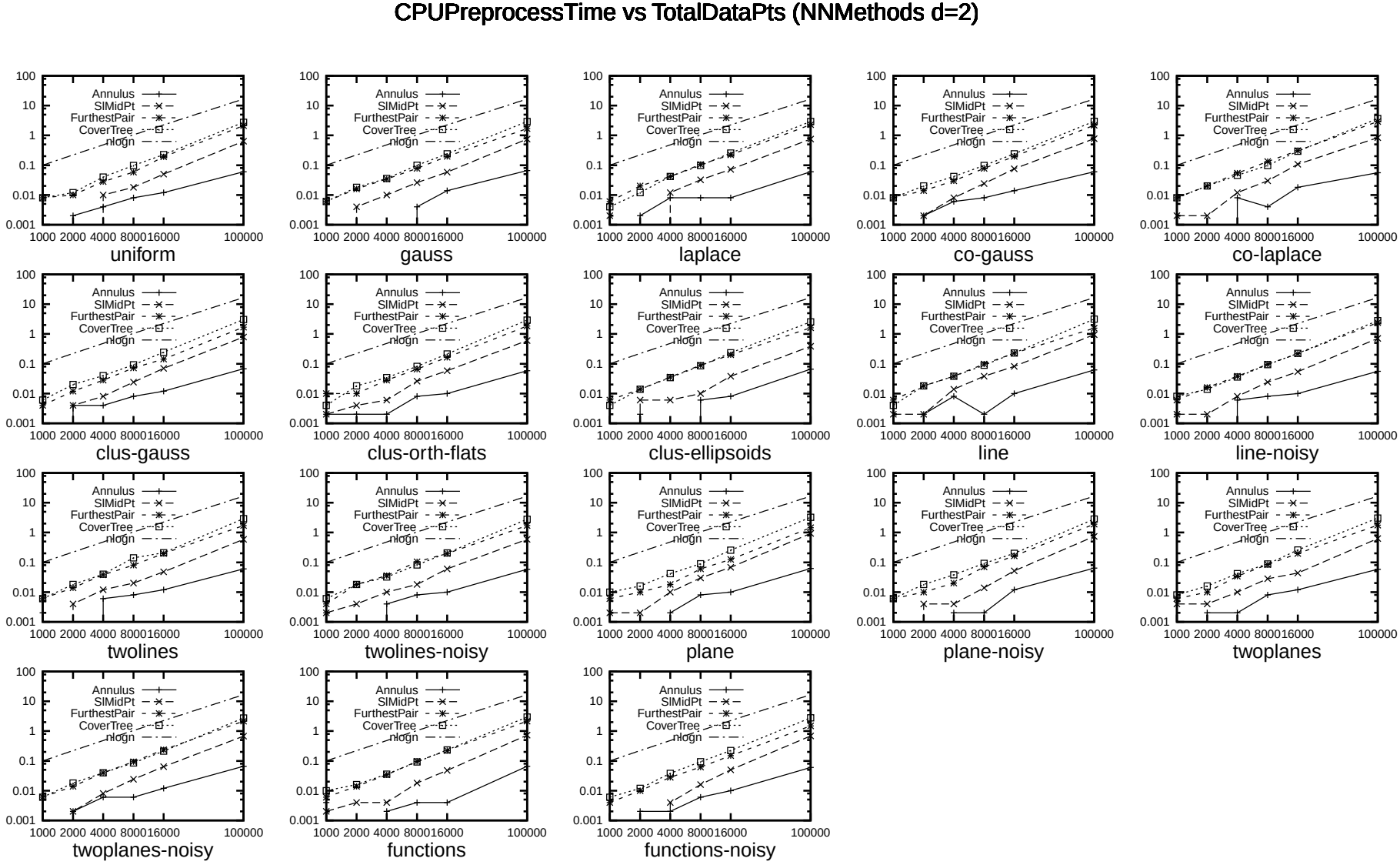


Figure C.1: Preprocessing time of NN methods for increasing n with d=2.

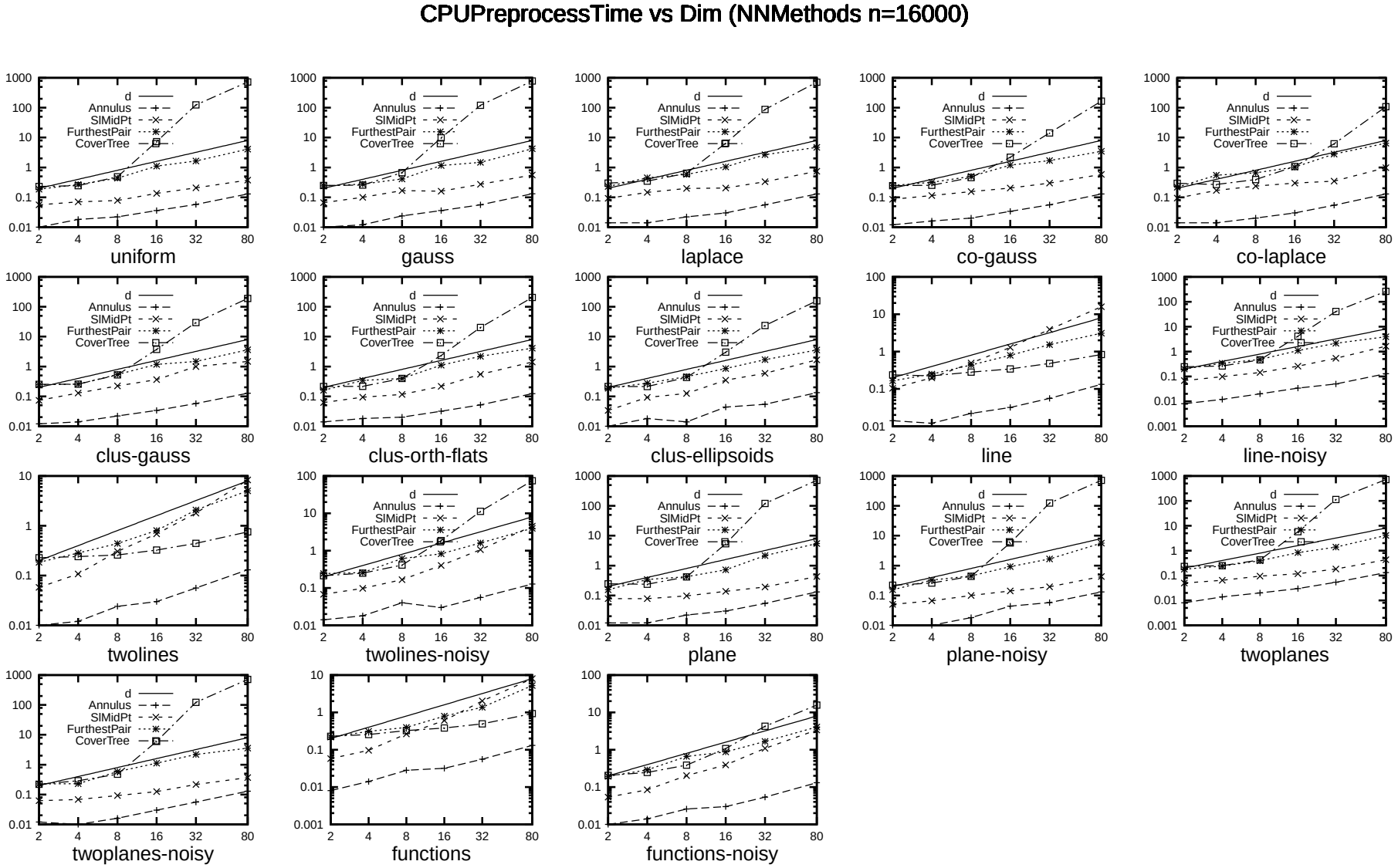


Figure C.2: Preprocessing time of NN methods for increasing d with n=16K.

Bibliography

Aggarwal, A., Hansen, M. & Leighton, T. (1990). Solving query-retrieval problems by compacting voronoi diagrams. In *STOC '90: Proceedings of the twenty-second annual ACM symposium on Theory of computing* (pp. 331–340). New York, NY, USA: ACM Press.

Aggarwal, C. C. (2002). On effective classification of strings with wavelets. In *KDD '02: Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining* (pp. 163–172). New York, NY, USA: ACM Press.

Andoni, A. & Indyk, P. (2006). Efficient algorithms for substring near neighbor problem. In *SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm* (pp. 1203–1212). New York, NY, USA: ACM Press.

Arge, L., de Berg, M., Haverkort, H. J. & Yi, K. (2004). The priority r-tree: a practically efficient and worst-case optimal r-tree. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data* (pp. 347–358). New York, NY, USA: ACM Press.

Arya, S., Malamatos, T. & Mount, D. M. (2002). Space-efficient approximate voronoi diagrams. In *STOC '02: Proceedings of the thiry-fourth annual ACM symposium on Theory of computing* (pp. 721–730). New York, NY, USA: ACM Press.

Arya, S. & Mount, D. M. (1993). Algorithms for fast vector quantization. In *Proceedings of the IEEE Data Compression Conference (DCC'93)* (pp. 381–390). IEEE Press.

Arya, S., Mount, D. M., Netanyahu, N. S., Silverman, R. & Wu, A. Y. (1998). An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *Journal of the ACM*, *45(6)*, 891–923.

Atkeson, C. G., Moore, A. W. & Schaal, S. (1997). Locally weighted learning. *Artificial Intelligence Review*, *11(1-5)*, 11–73.

Aurenhammer, F. (1991). Voronoi diagrams–A survey of a fundamental geometric data structure. *ACM Computing Surveys*, *23(3)*, 345–405.

Bach, J. R., Fuller, C., Gupta, A., Hampapur, A., Horowitz, B., Humphrey, R., Jain, R. C. & Shu, C.-F. (1996). Virage image search engine: an open framework for image management. In Sethi, I. K. & Jain, R. C. (Eds.), *Proceedings of SPIE Vol. 2670*, Volume 2670 (pp. 76–87). SPIE.

Beckmann, N., Kriegel, H.-P., Schneider, R. & Seeger, B. (1990). The r*-tree: an efficient and robust access method for points and rectangles. In *SIGMOD '90: Proceedings of the 1990 ACM SIGMOD international conference on Management of data* (pp. 322–331). New York, NY, USA: ACM Press.

Bei, C.-D. & Gray, R. M. (1985). An improvement of the minimum distortion encoding algorithm for vector quantization. *IEEE Transactions on Communications*, *COM-33(10)*, 1132–1133.

Beis, J. S. & Lowe, D. G. (1997). Shape indexing using approximate nearest-neighbour search in high-dimensional spaces. In *CVPR '97: Proceedings of the 1997 Conference on Computer Vision and Pattern Recognition (CVPR '97)* (p. 1000). Washington, DC, USA: IEEE Computer Society.

Bentley, J. L. (1975). Multidimensional binary search trees used for associative searching. *Commun. ACM*, *18(9)*, 509–517.

Berchtold, S., Keim, D. A. & Kriegel, H.-P. (1996). The x-tree : An index structure for high-dimensional data. In Vijayaraman, T. M., Buchmann, A. P., Mohan, C. & Sarda, N. L. (Eds.), *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India* (pp. 28–39). Morgan Kaufmann.

Beyer, K. S., Goldstein, J., Ramakrishnan, R. & Shaft, U. (1999). When is "nearest neighbor" meaningful? In *ICDT '99: Proceeding of the 7th International Conference on Database Theory* (pp. 217–235). London, UK: Springer-Verlag.

Beygelzimer, A., Kakade, S. & Langford, J. (2005). Cover trees for nearest neighbor. Unpublished. http://www.hunch.net/∼jl/projects/cover_tree/paper/paper.ps.

Beygelzimer, A., Kakade, S. & Langford, J. (2006). Cover trees for nearest neighbor. In *ICML '06: Proceedings of the 23rd international conference on Machine learning* (pp. 97–104). New York, NY, USA: ACM Press.

Brin, S. (1995). Near neighbor search in large metric spaces. In *VLDB '95: Proceedings of the 21th International Conference on Very Large Data Bases* (pp. 574–584). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Cai, Y., de Freitas, N. & Little, J. J. (2006). Robust visual tracking for multiple targets. In Leonardis, A., Bischof, H. & Pinz, A. (Eds.), *Computer Vision – ECCV06 9th European Conference on Computer Vision, Graz, Austria, May 7-13, 2006, Proceedings, Part IV*, Volume 3954 of *Lecture Notes in Computer Science* (pp. IV: 107–118). Springer.

Ciaccia, P., Patella, M. & Zezula, P. (1997). M-tree: An efficient access method for similarity search in metric spaces. In *VLDB '97: Proceedings of the 23rd International Conference on Very Large Data Bases* (pp. 426–435). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Clarkson, K. L. (1999). Nearest neighbor queries in metric spaces. *Discrete & Computational Geometry*, *22(1)*, 63–93.

Clarkson, K. L. (2002). Nearest neighbor searching in metric spaces: Experimental results for sb(S). See http://cm.bell-labs.com/who/clarkson/Msb/white_paper.pdf and http://cm.bell-labs.com/who/clarkson/Msb/readme.html.

Cover, T. M. & Hart, P. E. (1967). Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, *13(1)*, 21–27.

Datar, M., Immorlica, N., Indyk, P. & Mirrokni, V. S. (2004). Locality-sensitive hashing scheme based on p-stable distributions. In *SCG '04: Proceedings of the twentieth annual symposium on Computational geometry* (pp. 253–262). New York, NY, USA: ACM Press.

Deerwester, S. C., Dumais, S. T., Landauer, T. K., Furnas, G. W. & Harshman, R. A. (1990). Indexing by latent semantic analysis. *Journal of the American Society of Information Science*, *41(6)*, 391–407.

Deng, K. & Moore, A. (1995). Multiresolution instance-based learning. In *Proceedings of the Twelfth International Joint Conference on Artificial Intellingence* (pp. 1233–1239). San Francisco: Morgan Kaufmann.

Dudoit, S., Fridlyand, J. & Speed, T. P. (2002). Comparison of discrimination methods for the classification of tumors using gene expression data. *Journal of the American Statistical Association*, *97(457)*, 77–87.

Fagin, R. & Stockmeyer, L. (1998). Relaxing the triangle inequality in pattern matching. *International Journal of Computer Vision*, *28(3)*, 219–231.

Faloutsos, C. & Oard, D. W. (1995). A survey of information retrieval and filtering methods. Technical Report CS-TR-3514, Department of Computer Science, University of Maryland.

Faragó, A., Linder, T. & Lugosi, G. (1993). Fast nearest-neighbor search in dissimilarity spaces. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *15(9)*, 957–962.

Fayyad, U. M., Piatetsky-Shapiro, G., Smyth, P. & Uthurusamy, R. (1996). *Advances in Knowledge Discovery and Data Mining*, chapter 4, (p. 102). The MIT Press.

Flickner, M., Sawhney, H., Niblack, W., Ashley, J., Huang, Q., Dom, B., Gorkani, M., Hafner, J., Lee, D., Petkovic, D., Steele, D. & Yanker, P. (1995). Query by image and video content: The qbic system. *Computer*, *28(9)*, 23–32.

Friedman, J. H., Bentley, J. L. & Finkel, R. A. (1977). An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematics Software*, *3(3)*, 209–226.

Gersho, A. & Gray, R. M. (1991). *Vector quantization and signal compression* (1 Ed.). Norwell, MA, USA: Kluwer Academic Publishers.

Ghias, A., Logan, J., Chamberlin, D. & Smith, B. C. (1995). Query by humming: musical information retrieval in an audio database. In *MULTIMEDIA '95: Proceedings of the third ACM international conference on Multimedia* (pp. 231–236). New York, NY, USA: ACM Press.

Gionis, A., Indyk, P. & Motwani, R. (1999). Similarity search in high dimensions via hashing. In Atkinson, M. P., Orlowska, M. E., Valduriez, P., Zdonik, S. B. & Brodie, M. L. (Eds.), *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK* (pp. 518–529). Morgan Kaufmann.

Gray, A. & Moore, A. (2004). Tutorial on Data Structures for Fast Statistics. In 21st International Conference on Machine Learning, 2004. Available from http://www.cs.cmu.edu/∼agray/icml.html.

Gupta, A., Krauthgamer, R. & Lee, J. R. (2003). Bounded geometries, fractals, and low-distortion embeddings. In *FOCS '03: Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science* (p. 534). Washington, DC, USA: IEEE Computer Society.

Guttman, A. (1984). R-trees: a dynamic index structure for spatial searching. In *SIGMOD '84: Proceedings of the 1984 ACM SIGMOD international conference on Management of data* (pp. 47–57). New York, NY, USA: ACM Press.

Hastie, T., Tibshirani, R. & Friedman, J. (2001). *The Elements of Statistical Learning Data Mining, Inference, and Prediction*, chapter 2, (pp. 22–27). Springer Series in Statistics. Springer.

Hinneburg, A., Aggarwal, C. C. & Keim, D. A. (2000). What is the nearest neighbor in high dimensional spaces? In *VLDB '00: Proceedings of the 26th International Conference on Very Large Data Bases* (pp. 506–515). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Huang, C.-M., Q. Bi, G. S. S. & Harris, R. W. (1992). Fast full search equivalent encoding algorithms for image compression using vector quantization. *IEEE Transactions on Image Processing*, *1(3)*, 413–416.

Indyk, P. (1998). On approximate nearest neighbors in non-euclidean spaces. In *FOCS '98: Proceedings of the 39th Annual Symposium on Foundations of Computer Science* (p. 148). Washington, DC, USA: IEEE Computer Society.

Indyk, P. (2002). Approximate nearest neighbor algorithms for frechet distance via product metrics. In *SCG '02: Proceedings of the eighteenth annual symposium on Computational geometry* (pp. 102–106). New York, NY, USA: ACM Press.

Indyk, P. & Motwani, R. (1998). Approximate nearest neighbors: towards removing the curse of dimensionality. In *STOC '98: Proceedings of the thirtieth annual ACM symposium on Theory of computing* (pp. 604–613). New York, NY, USA: ACM Press.

Karger, D. R. & Ruhl, M. (2002). Finding nearest neighbors in growth-restricted metrics. In *STOC '02: Proceedings of the thiry-fourth annual ACM symposium on Theory of computing* (pp. 741–750). New York, NY, USA: ACM Press.

Katayama, N. & Satoh, S. (1997). The sr-tree: An index structure for high-dimensional nearest neighbor queries. In Peckham, J. (Ed.), *SIGMOD 1997, Proceedings ACM*

SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA (pp. 369–380). ACM Press.

Katayama, N. & Satoh, S. (2001). Distinctiveness-sensitive nearest-neighbor search for efficient similarity retrieval of multimedia information. In *ICDE '01: Proceedings of the 17th International Conference on Data Engineering* (p. 493). Washington, DC, USA: IEEE Computer Society.

Knuth, D. E. (1997). *The art of computer programming, volume 2: Seminumerical Algorithms* (3 Ed.). Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc.

Krauthgamer, R. & Lee, J. R. (2004). Navigating nets: simple algorithms for proximity search. In *SODA '04: Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms* (pp. 798–807). Philadelphia, PA, USA: Society for Industrial and Applied Mathematics.

Lee, D.-T. (1982). On k-nearest neighbor voronoi diagrams in the plane. *IEEE Transactions on Computers*, *C-31(6)*, 478–487.

Li, T., Zhang, C. & Ogihara, M. (2004). A comparative study of feature selection and multiclass classification methods for tissue classification based on gene expression. *Bioinformatics*, *20(15)*, 2429–2437.

Lin, K.-I., Jagadish, H. V. & Faloutsos, C. (1994). The tv-tree: An index structure for high-dimensional data. *VLDB J.*, *3(4)*, 517–542.

Liu, T., Moore, A. W., Gray, A. & Yang, K. (2005). An investigation of practical approximate nearest neighbor algorithms. In L. K. Saul, Y. Weiss & L. Bottou (Eds.), *Advances in Neural Information Processing Systems 17* (pp. 825–832). Cambridge, MA: MIT Press.

Liu, T., Moore, A. W. & Gray, A. G. (2004). Efficient exact k-nn and nonparametric classification in high dimensions. In *Advances in Neural Information Processing Systems 16*. Cambridge, MA: MIT Press.

Lucarella, D. (1988). A document retrieval system based on nearest neighbour searching. *Journal of Information Science*, *14(1)*, 25–33.

Manber, U. (1989). *Introduction to Algorithms: A Creative Approach*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Maneewongvatana, S. & Mount, D. M. (1999). It's okay to be skinny, if your friends are fat.

Maneewongvatana, S. & Mount, D. M. (2001). On the efficiency of nearest neighbor searching with data clustered in lower dimensions. Technical Report UMIACS-TR-2001-05, Univ. of Maryland, College Park, Inst. for Advanced Computer Studies.

Maneewongvatana, S. & Mount, D. M. (2002). Analysis of approximate nearest neighbor searching with clustered point sets. In Goldwasser, M. H. & McGeoch, C. C. (Eds.), *Data Structures, Near Neighbor Searches, and Methodology: Fifth and Sixth DIMACS Implementation Challenges*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science (pp. 105–123). American Mathematical Society.

Marshall, D. (2006). Nearest neighbour searching in highdimensional metric space. MSc subthesis, Australian National University.

McNab, R. J., Smith, L. A., Bainbridge, D. & Witten, I. H. (1997). The new zealand digital library melody index. *D-Lib Magazine*, *3(5)*, 22–32.

Micó, M. L., Oncina, J. & Vidal, E. (1994). A new version of the nearest-neighbour approximating and eliminating search algorithm (AESA) with linear preprocessing time and memory requirements. *Pattern Recognition Letters*, *15(1)*, 9–17.

Minsky, M. & Papert, S. (1969). *Perceptrons* (pp. 222–225). MIT Press.

Mollineda, R. A., Vidal, E. & Martínez-Hinarejos, C. D. (2003). Adaptive learning for string classification. In *IbPRIA* (pp. 564–571). Springer.

Moore, A. (1991). A tutorial on kd-trees. Extract from PhD Thesis. Available from http://www.autonlab.org/autonweb/14665.html.

Moore, A., Schneider, J. & Deng, K. (1997). Efficient locally weighted polynomial regression predictions. In Fisher, D. (Ed.), *Proceedings of the Fourteenth International Conference on Machine Learning* (pp. 236–244). 340 Pine Street, 6th Fl., San Francisco, CA 94104: Morgan Kaufmann.

Moore, A. W. (2000). The anchors hierarchy: Using the triangle inequality to survive high dimensional data. In *UAI '00: Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence* (pp. 397–405). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Mount, D. M. (2006). *ANN Programming Manual* (1.1.1 Ed.). College Park, MD, USA: Department of Computer Science, University of Maryland.

Mount, D. M. & Arya, S. (1997). ANN: A library for approximate nearest neighbor searching. In CGC 2nd Annual Fall Workship on Computational Geometry, 1997. Available from http://www.cs.umd.edu/∼mount/ANN.

Niijima, S. & Kuhara, S. (2005). Effective nearest neighbor methods for multiclass cancer classification using microarray data. Poster presented at the 16th International Conference on Genome Informatics, December 19-21, 2005, Yokohama Pacifico, Japan. Available from http://www.jsbi.org/journal/GIW05/GIW05P051.pdf.

Okabe, A., Boots, B., Sugihara, K. & Chiu, S. N. (2000). *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams* (2 Ed.). John Wiley & Sons.

Omohundro, S. M. (1989). Five balltree construction algorithms. Technical Report TR-89-063, International Computer Science Institute.

Orchard, M. T. (1991). A fast nearest-neighbor search algorithm. In *1991 International Conference on Acoustics, Speech and Signal Processing, 1991. ICASSP-91.*, Volume 4 (pp. 2297–3000). IEEE Press.

Pentland, A. P., Picard, R. W. & Scarloff, S. (1994). Photobook: tools for content-based manipulation of image databases. In Niblack, C. W. & Jain, R. C. (Eds.), *Proceedings of SPIE Vol. 2185*, Volume 2185 (pp. 34–47). SPIE.

Press, W. H., Teukolsky, S. A., Vetterling, W. T. & Flannery, B. P. (1992). *Numerical Recipes in C: The Art of Scientific Computing* (2 Ed.). New York, NY, USA: Cambridge University Press.

Robinson-Cox, J. F., Bateson, M. M. & Ward, D. M. (1995). Evaluation of nearest-neighbor methods for detection of chimeric small-subunit rrna sequences. *Applied and Environmental Microbiology, 61(4)*, 1240–1245.

Sellis, T. K., Roussopoulos, N. & Faloutsos, C. (1987). The r+-tree: A dynamic index for multi-dimensional objects. In Stocker, P. M., Kent, W. & Hammersley, P. (Eds.), *VLDB'87, Proceedings of 13th International Conference on Very Large Data Bases, September 1-4, 1987, Brighton, England* (pp. 507–518). Morgan Kaufmann.

Shakhnarovich, G., Darrell, T. & Indyk, P. (2006a). *Nearest-Neighbor Methods in Learning and Vision: Theory and Practice*, chapter 1, (pp. 1–2). The MIT Press.

Shakhnarovich, G., Darrell, T. & Indyk, P. (2006b). *Nearest-Neighbor Methods in Learning and Vision: Theory and Practice* (1 Ed.) (p.6̃). Cambridge, MA: The MIT Press.

Shakhnarovich, G., Darrell, T. & Indyk, P. (2006c). *Nearest-Neighbor Methods in Learning and Vision: Theory and Practice* (1 Ed.) (p.6̃4). Cambridge, MA: The MIT Press.

Smith, J. R. & Chang, S.-F. (1996). Visualseek: a fully automated content-based image query system. In *MULTIMEDIA '96: Proceedings of the fourth ACM international conference on Multimedia* (pp. 87–98). New York, NY, USA: ACM Press.

Sproull, R. F. (1991). Refinements to nearest-neighbor searching in k-dimensional trees. *Algorithmica*, *6(4)*, 579–589.

Talbert, D. A. & Fisher, D. (2000). An empirical analysis of techniques for constructing and searching k-dimensional trees. In *KDD '00: Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining* (pp. 26–33). New York, NY, USA: ACM Press.

Tseng, Y.-H. (1999). Content-based retrieval for music collections. In *SIGIR '99: Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval* (pp. 176–182). New York, NY, USA: ACM Press.

Uhlmann, J. K. (1991a). Metric trees. *Applied Mathematics Letters*, *4(5)*, 61–62.

Uhlmann, J. K. (1991b). Satisfying general proximity / similarity queries with metric trees. *Information Processing Letters*, *40(4)*, 175–179.

Uitdenbogerd, A. L. & Zobel, J. (2002). Music ranking techniques evaluated. In *ACSC '02: Proceedings of the twenty-fifth Australasian conference on Computer science* (pp. 275–283). Darlinghurst, Australia, Australia: Australian Computer Society, Inc.

Vidal, E. (1986). An algorithm for finding nearest neighbours in (approximately) constant average time. *Pattern Recognition Letters*, *4(3)*, 145–157.

Vidal, E., Rulot, H. M., Casacuberta, F. & Benedí, J.-M. (1988). On the use of a metric-space search algorithm (aesa) for fast dtw-based recognition of isolated words. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, *36(5)*, 651–660.

Weber, R. & Blott, S. (1997). An approximation based data structure for similarity search. Technical report, Institute of Information Systems, ETH Zurich.

Weber, R., Schek, H.-J. & Blott, S. (1998). A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB '98: Proceedings of the 24rd International Conference on Very Large Data Bases* (pp. 194–205). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

White, D. A. & Jain, R. (1996). Similarity indexing with the ss-tree. In *ICDE '96: Proceedings of the Twelfth International Conference on Data Engineering* (pp. 516–523). Washington, DC, USA: IEEE Computer Society.

Witten, I. H. & Frank, E. (2005). *Data Mining: Practical machine learning tools and techniques* (2 Ed.). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Yianilos, P. N. (1993). Data structures and algorithms for nearest neighbor search in general metric spaces. In *SODA '93: Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms* (pp. 311–321). Philadelphia, PA, USA: Society for Industrial and Applied Mathematics.

Zatloukal, K., Johnson, M. H. & Ladner, R. E. (2002). Nearest neighbor search for data compression. In Goldwasser, M. H. & McGeoch, C. C. (Eds.), *Data Structures, Near Neighbor Searches, and Methodology: Fifth and Sixth DIMACS Implementation Challenges*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science (pp. 69–87). American Mathematical Society.

Zhu, Y., Shasha, D. & Zhao, X. (2003). Query by humming: in action with its technology revealed. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data* (pp. 675–675). New York, NY, USA: ACM Press.