Proceedings of the Twenty-Seventh International Florida Artificial Intelligence Research Society Conference

Mitigating the Curse of Dimensionality for Exact *k*NN Retrieval

Michael A. Schuh
Dept. of Computer Science
Montana State University
Bozeman, MT, 59717 USA

Tim Wylie
Dept. of Computer Science
University of Alberta
Edmonton, AB, T6G2E8 Canada

Rafal A. Angryk
Dept. of Computer Science
Georgia State University
Atlanta, GA, 30302 USA

Abstract

Efficient data indexing and exact k-nearest-neighbor (kNN) retrieval are still challenging tasks in high-dimensional spaces. This work highlights the difficulties of indexing in high-dimensional and tightly-clustered dataspaces by exploring several important tunable parameters for optimizing kNN query performance using the iDistance and iDStar algorithms. We experiment on real and synthetic datasets of varying size, cluster density, and dimensionality, and compare performance primarily through filter-and-refine efficiency and execution time. Results show great variability over parameter values and provide new insights and justifications in support of prior best-use practices. Local segmentation with iDStar consistently outperforms iDistance in any clustered space below 256 dimensions, setting a new benchmark for efficient and exact kNN retrieval in high-dimensional spaces. We propose several directions of future work to further increase performance in high-dimensional real-world settings.

1. Introduction

Massive databases composed of rich information are becoming ubiquitous in the modern world. While storage of data is becoming routine, efficiently indexing and retrieving it is still an important practical concern. A frequent and costly information retrieval and data mining task performed on these databases is k-nearest neighbor (kNN) search, which returns the k most similar records to any given query point. While a database management system (DBMS) is highly optimized for a few dimensions, most traditional indexing algorithms (*e.g.*, the B-tree and R-tree families) degrade quickly as the number of dimensions increase. Often in these circumstances, the most efficient retrieval method available is a sequential (linear) scan of every single record in the database.

Many algorithms have been proposed in the past with limited success for true high-dimensional indexing, and this general problem is commonly referred to as *the curse of dimensionality* (Bellman 1957). In practice, this issue is often mitigated by applying dimensionality reduction techniques before using popular multi-dimensional indexing methods, and sometimes even by adding application logic to combine multiple independent indices or requiring user involvement during search. However, modern applications are increasingly employing highly-dimensional techniques to effectively represent massive data. For example, 128-dimensional SIFT features (Lowe 1999) are quite popular in Content-Based Image Retrieval (CBIR) systems where it is of critical importance to be able to comprehensively index all dimensions for a unified similarity-based retrieval model.

This work builds upon previous analyses of iDistance partitioning strategies and iDStar extensions (Schuh et al. 2013; Wylie et al. 2013; Schuh, Wylie, and Angryk 2013), with the continued goal of increasing overall performance of indexing and retrieval for kNN queries in high-dimensional and tightly-clustered dataspaces. Here we address the differences between these two types of dataspaces through the analysis of several tunable parameters. We assess performance efficiency by the total and accessed number of B$^+$-tree nodes, number of candidate data points returned from filtering, and the time taken to build the index and perform queries. In combination, these metrics provide a highly descriptive and unbiased quantitative benchmark for independent comparative evaluations.

The rest of the paper is organized as follows. Section 2 provides a brief background of related works, followed by an overview of iDistance and iDStar in Section 3. We present experimental results in Section 4 and close with conclusions and future work in Section 5.

2. Related Work

The ability to efficiently index and retrieve data has become a silent backbone of modern society, and it defines the capabilities and limitations of practical data usage. While the one-dimensional B$^+$-tree (Bayer and McCreight 1972) is foundational to the modern relational DBMS, most real data has many dimensions that require efficient access. Mathematics has long-studied the partitioning of multi-dimensional metric spaces, most notably Voronoi Diagrams and the related Delaunay triangulations (Aurenhammer 1991), but these theoretical solutions can often be too complex for practical application. To address this issue, many approximate techniques have been proposed for practical use. One of the most popular is the R-tree (Guttman 1984), which was developed with minimum bounding rectangles (MBRs) to build a hierarchical tree of successively

Copyright © 2014, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

smaller MBRs containing objects in a multi-dimensional space. The R*-tree (Beckmann et al. 1990) enhanced search efficiency by minimizing MBR overlap. However, these trees (and most derivations) quickly degrade in performance as the number of dimensions increases (Berchtold, Böhm, and Kriegal 1998; Ooi et al. 2000).

Research has recently focused on creating indexing methods that define a one-way lossy mapping function from a multi-dimensional space to a one-dimensional space that can then be indexed efficiently in a standard B$^+$-tree. These lossy mappings require a filter-and-refine strategy to produce exact query results, where the one-dimensional index is used to quickly retrieve a subset of the data points as candidates (the filter step), and then each of these candidates is verified to be within the specified query region in the original multi-dimensional space (the refine step). Since checking the candidates in the actual dataspace is costly, the goal of the filter step is to return as few candidates as possible while retaining the exact results to satisfy the query.

First published in 2001, iDistance (Yu et al. 2001; Jagadish et al. 2005) specifically addressed exact *k*NN queries in high-dimensional spaces and has proven to be one of the most efficient state-of-the-art techniques available. In recent years, iDistance has been used in a number of demanding applications, including: large-scale image retrieval (Zhang et al. 2006), video indexing (Shen, Ooi, and Zhou 2005), mobile computing (Ilarri, Mena, and Illarramendi 2006), peer-to-peer systems (Doulkeridis et al. 2007), and video surveillance retrieval (Qu, Chen, and Yang 2008). As information retrieval from increasingly high-dimensional and large-scale databases continues to grow in need and ubiquity, the motivations for furthering this research are clearly present. While many recent works have shifted focus to approximate nearest neighbor techniques (Indyk and Motwani 1998; Tao et al. 2009) that can satisfy some of these applications, in general they are outside the scope of efficient exact *k*NN retrieval presented in this paper.

3. Algorithms

We begin with a brief overview of the original iDistance algorithm and explain the basic index and query mechanisms. These are also the foundations of iDStar, a new hybrid index that incorporates heuristic-guided spatial segmentation to better prune congested areas of the dataspace.

iDistance

The basic concept of iDistance is to segment the dataspace into disjoint spherical partitions, where all points in a partition are *indexed by their distance* to the reference point of that partition. This results in a set of one-dimensional distance values for each partition, where each distance is related to one or more data points within that partition. The algorithm was motivated by the ability to use arbitrary reference points to determine the similarity between any two data points in a metric space, allowing single dimensional ranking and indexing of data despite the original dimensionality (Yu et al. 2001; Jagadish et al. 2005).

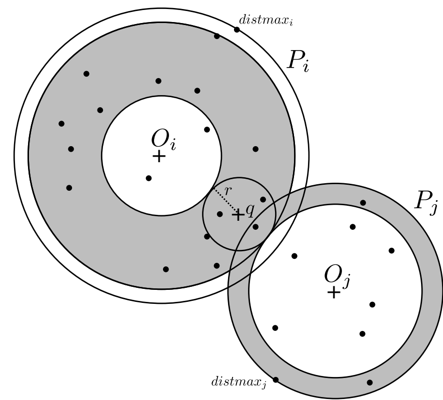

Figure 1: A query sphere q with radius r and the searched regions (shaded) in the two overlapping partitions P_i and P_j defined by their reference points O_i and O_j, and radii $distmax_i$ and $distmax_j$, respectively.

Building the Index The index is best built with *data-based* partitioning strategies, which adjust the size and location of partitions in the dataspace according to the underlying data distribution, which greatly increases retrieval performance in real-world settings (Yu et al. 2001; Jagadish et al. 2005; Schuh et al. 2013). The alternative is *space-based* partitioning, which statically partitions the dataspace without information about the data, but these strategies are found to be highly susceptible to dimensionality (Schuh et al. 2013). For all partitioning strategies, data points are assigned to the single closest partition based on Euclidean distance to each partition's representative reference point.

Formally, we have a set of partitions $\mathbb{P} = \langle P_1, \ldots, P_M \rangle$ with respective reference points $\mathbb{O} = \langle O_1, \ldots, O_M \rangle$. We will let the number of points in a partition be denoted $|P_i|$ and the total number of points in the dataset be N. After the partitions are defined, a mapping function is applied to create separation in the underlying B$^+$-tree between each partition, ensuring that any given index value represents a unique distance for exactly one partition. This is a lossy mapping function, which is mathematically surjective, but not injective, so all points equi-distant from the reference point will map to the same one-dimensional value.

The index value y_p for a point p assigned to a partition P_i with reference point O_i is given by Equation 1, where $dist()$ is any metric distance function, i is the partition index, and c is a constant multiplier for creating the partition separation. While constructing the index, each partition P_i records the distance of its farthest point as $distmax_i$. We can safely and automatically set $c = 2\sqrt{d}$, which is twice the maximum possible distance of two points in a d-dimensional unit dataspace, and therefore no index value in partition P_i will clash with values in any other partition $P_{j \neq i}$.

$$y_p = i \times c + dist(O_i, p) \tag{1}$$

Querying the Index The index should be built in such a way that the filter step of a query returns the fewest possible candidate points while retaining the true k-nearest neigh-

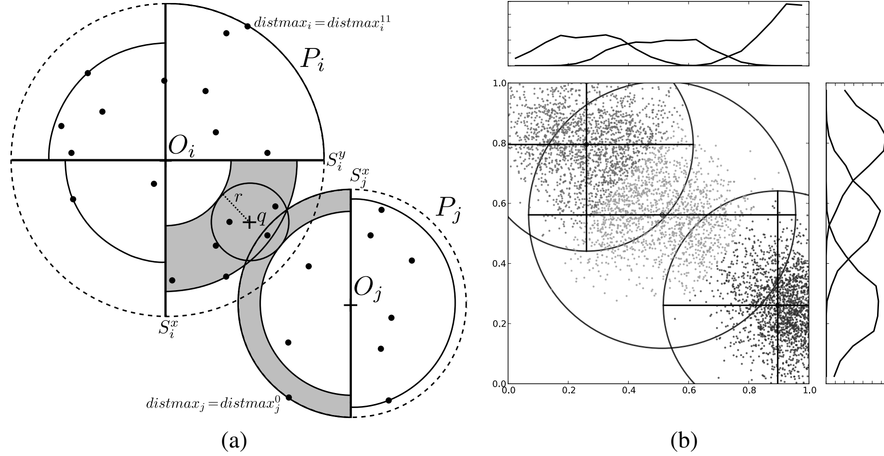

Figure 2: (a) Conceptual local splits S_i^x, S_i^y, and S_j^x and their effect on query search ranges in overlapping partitions and sections. (b) The local population-based splits applied to each partition by the L3 heuristic on the example dataset.

bors. Fewer candidates reduce the costly refinement step which must verify the true multi-dimensional distance of each candidate from the query point. Performing a query q with radius r consists of three steps: 1) determine the set of partitions to search, 2) calculate the search range of each partition in the set, and 3) retrieve the candidate points and refine by their true distances. The query is finished when it can guarantee that no other point yet to be retrieved can be closer to q than the farthest point already in the k-set.

Figure 1 shows an example query sphere contained completely within partition P_i and intersecting partition P_j, as well as the shaded ranges of each partition that need to be searched. For each partition P_i and its $distmax_i$, the query sphere overlaps the partition if the distance from the edge of the query sphere to the reference point O_i is less than $distmax_i$, as defined in Equation 2. There are two possible cases of overlap: 1) q resides within P_i, or 2) q is outside of P_i, but the query sphere intersects it. In the first case, the partition needs to be searched both inward and outward from the query point over the range $(q \pm r)$, whereas in the second case, a partition only needs to be searched inward from the edge $(distmax_i)$ to the closest point of query intersection.

$$dist(O_i, q) - r \leq distmax_i \tag{2}$$

iDStar

In previous work, it was shown that in high-dimensional and tightly-clustered spaces, iDistance stabilizes in performance by accessing an entire partition (k-means derived cluster) to satisfy a given query (Schuh et al. 2013). While only accessing a single partition is already significantly more efficient than sequential scan, this weakness was the main motivation to explore further dataspace segmentation extensions to enhance retrieval performance in these spaces. These extensions are similar in concept to the works of the iMinMax(θ) (Ooi et al. 2000) and SIMP (Singh and Singh 2012) algorithms, whereby we can incorporate additional dataspace knowledge at the price of added algorithmic complexity and performance overhead. Recently published work introduced iDStar and proved the feasibility of our first heuristic-guided extensions (Schuh, Wylie, and Angryk 2013).

For official vernacular, it can be said we further separate dense areas of the dataspace by *splitting* partitions into disjoint *sections* corresponding to separate *segments* of the B$^+$-tree that can be selectively pruned during retrieval. In other words, we apply a number of dimensional *splits* in the dataspace which *sections* the partitions into B$^+$-tree *segments*.

We focus our discussion on our local segmentation extension, which explicitly splits the dataspace of each partition separately. To achieve this, several modifications to the indexing and retrieval algorithms are required, which implicitly affect our use of the underlying B$^+$-tree . First, the mapping function is updated to create a constant segment separation within the already separated partitions. Equation 3 describes the new function with the inclusion of the sectional index j, and s as the total number of splits applied to the partition. Note that a partition with s splits is divided into 2^s sections, so we must be careful to appropriately bound the number of splits applied. Second, after identifying the partitions to search, we must identify the sections within each partition to actually search, as well as their updated search ranges within the B$^+$-tree. This also requires the overhead of section-supporting data structures in addition to the existing partition-supporting data structures, but this extra space cost is negligible compared to the added time costs.

$$y_p = i \times c + j \times \frac{c}{2^s} + dist(O_i, p) \tag{3}$$

The local technique is a purposeful partition-specific segmentation method based only on local information of the given partition rather than the global data characteristics of the entire space. Local splits are defined by a tuple of partition and dimension, whereby the given partition is split in the specified dimension, and all splits are made directly on the partition reference point value. Figure 2(a) shows a conceptual example of local splits.

During index creation we now also maintain a sectional $distmax_i^j$ (for partition P_i and section j). This allows the addition of a section-based query overlap filter which can often prune away entire partitions that would have otherwise been searched. This can be seen in Figure 2(a), where the original partition $distmax$ is now a dotted line and a new $distmax$ is shown for each individual partition section.

Within each partition, we further split the data as evenly as possible to maximize the amount of separation in the B$^+$-tree by creating more unique index values. Since splits are made at the reference points, we collect additional spatial statistics from each to rank the dimensions by those closest to an equal (50/50) population split. We denote s as the maximum number of splits any partition may have. Our initial heuristic uniformly applied s splits to each partition using the top ranked dimensions for each. However, if the population is already small enough, it may not be beneficial to split the partition at all. Therefore, we designed an automatic method for optimal population analysis. For this heuristic (L3), we first use s to calculate the maximum number of underlying B$^+$-tree segments that would be uniformly applied, and then we distribute them based on relative partition population. This automatically applies additional splits to partitions in dense areas of the dataspace by taking splits away from other partitions where they are not needed. Most importantly, it still maintains the upper-bound on B$^+$-tree segments and resultant tree depth.

Formally, we have a nested set of splits $\mathbb{S} = \langle S_1 \ldots, S_M \rangle$ where S_i represents the set of dimensions to split on for partition P_i. Given a maximum granularity for the leaves in the B$^+$-tree, we can calculate how many splits each partition should have as a percentage of its population to the total dataset size, shown in Equation 4. For example, we can see in Figure 2(b) that the center cluster did not meet the population percentage to have two splits. Thus, the only split is on the dimension that has the closest 50/50 population split.

$$|S_i| = \left\lfloor \lg \frac{|P_i|}{N} \cdot M \cdot 2^s \right\rfloor \tag{4}$$

The local dataspace segmentation generates a hybrid index with one-dimensional distance-based partitions subsequently segmented in a tune-able subset of dimensions. Due to the exponential growth of the tree segments, we must limit the number of dimensions we split on, which L3 maintains automatically. For the spatial subtree segmentation, we use z-curve ordering of each partition's sections, efficiently encoded as a ternary bitstring representation of all overlapping sections that is quickly decomposable to a list of section indices. Figure 2(b) shows an example of this ordering with $distmax_i^{11}$, which also happens to be the overall partition $distmax_i$. Since P_i is split twice, we have a two-digit binary encoding of quadrants, with the upper-right of '11' equating to section 3 (zero-based) of P_i.

Experiments and Results

Every experiment reports a set of statistics describing the index and query performance of that test. We primarily highlight three statistics from tested queries: 1) the number of candidate points returned during the filter step, 2) the number of nodes accessed in the B$^+$-tree, and 3) the time taken (in milliseconds) to perform the query and return the final results. Other descriptive statistics include the B$^+$-tree size (total nodes) and the number of dataspace partitions and sections (of partitions) that were checked during the query. We often express the ratio of candidates and nodes over the total number of points in the dataset and the total number of nodes in the B$^+$-tree, respectively, as this eliminates skewed results due to varying dataset characteristics. Note that all data fits in main memory, so all experiments are compared without costly I/O bottlenecks common in practice.

The first experiments are on synthetic datasets in a d-dimensional unit space $[0.0, 1.0]^d$. We provide the number of clusters and the standard deviation of the independent Gaussian distributions (in each dimension) for each randomly generated cluster center. For each dataset, we randomly select 500 points as *k*NN queries, which ensures that our query point distribution follows the dataset distribution.

Figure 3 illustrates the curse of dimensionality quite well through various pair-wise distance statistics on a synthetic clustered dataset. These are collected for every data point within the same partition, over all partitions of the dataset. In other words, we are explicitly looking at the distance between intra-cluster data points. Notice how the standard deviation (stddev) and relative average distance normalized by the maximum distance in the given dimensional space (relavg) remain essentially constant over dimensionality, but minimum and maximum distances are converging on the increasing average distance.

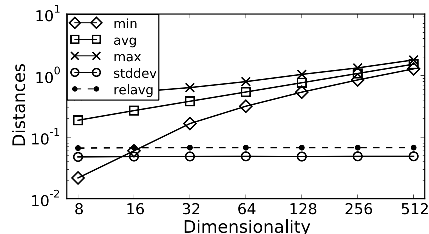

Figure 3: The curse of dimensionality through pair-wise intra-cluster distance over all partitions.

As dimensionality grows, the average distance between any two intra-cluster data points grows, but the distribution of these distances converges on everything becoming equidistant from each other. This is similar to tightly-clustered spaces, whereby all points become equidistant, however importantly, here the average distances decrease as cluster density (or compactness) increases.

Figure 4 shows the performance of iDStar local segmentation (L3) over varying dimensionality using 4, 8, and 12 splits determined by the L3 heuristic, compared to standard iDistance (TC) using the same underlying true cluster centers as reference points. Here we see a clear trade-off between the filtering power of local segmentation, and the performance overhead of adding additional splits. Under 256 dimensions, the additional filtering power not only works significantly better, but it also takes very little overhead –

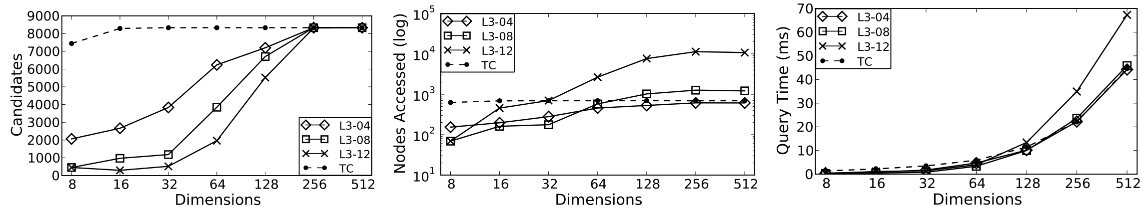

Figure 4: Comparing iDistance (TC) to iDStar (L3) with 4, 8, and 12 splits over dataset dimensionality.

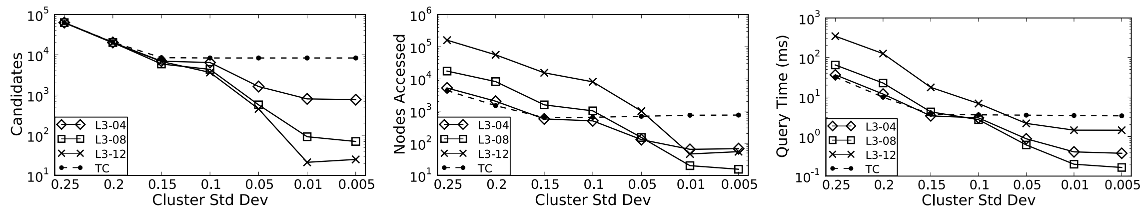

Figure 5: Comparing iDistance (TC) to iDStar (L3) with 4, 8, and 12 splits over dataset cluster standard deviation.

while above this there is little benefit in using it. One would hypothesize that even more splits would further mitigate the curse of dimensionality for filtering candidates, but the overhead cost in accessed nodes will dramatically increase and likely be impractical.

Figure 5 shows the performance of iDStar local segmentation (L3) in tightly-clustered data similarly using 4, 8, and 12 splits compared to iDistance (TC). While the distance-phenomenon is somewhat similar in tightly-clustered spaces, we can see a noticable difference as clusters become tighter. First notice that when clusters are quite spread out (0.25 *stdev* in each dimension) iDStar cannot filter the candidates any better than iDistance, but it suffers in time because of the overhead of additional nodes accessed. The real benefit is seen from 0.15 *stdev* and smaller, when proper segmentation eliminates many potential candidates, even though the space is extremely dense.

The difference between these two sets of charts exemplifies how difficult indexing high-dimensional spaces are compared to well-clustered, but congested areas. Consider again the conceptual examples of our spherical partitions in Figures 1 and 2(a). As we decrease standard deviation of these clusters, the partitions collapse inward to the reference point, where intra-cluster distances become equally small, but inter-cluster distances increase. On the other hand, as we increase dimensionality, these spherical partitions turn into ring-like bands of equally large distances that continue to expand away from the reference points. This leads to more overlapping of partitions and, given the equi-distant points, more candidates being returned from each search.

Next we look at how the number of partitions affects performance of iDistance (and also iDStar). Here we use a real world dataset of SIFT descriptors[1] consisting of 1 million points in 128 dimensions. Since these characteristics do not change, we tune performance by varying the number of partitions used, shown in Figure 6. The first chart shows that as we increase above 64 total partitions, queries access fewer and fewer partitions, primarily because more and more are empty. The "other" line represents partitions which are used (non-empty) but not searched.

The second chart in Figure 6 uses proportions to present four related statistics: 1) the percentage of candidates checked, the percentage of nodes accessed, the average fill of a node in the tree, and the relative amount of total nodes in the tree, based on the total node count for 1024 partitions. First, we see the expected results that more total partitions lead to fewer candidates and nodes accessed. However, we can see this reduction stagnates above 256 partitions, which is also where we see the structure of the tree change. Notice around 128 partitions, we see a reduction in total nodes, but an increase in average node fill, thereby indicating a more compact tree. This is a somewhat unexpected result, which might be caused in part by the constant partition separation and the increase in unique index values.

Lastly, the third chart in Figure 6 shows the time taken to complete three actions: 1) calculate the index values from the dataset, 2) build the tree of index values, and 3) perform a query. Here we use the relative values again (based on values at 1024 partitions) so we can compare all three side-by-side. Interestingly, the tree time is rather constant, but it does take somewhat longer when more partitions are used. This is likely caused by the more compact tree and possibly more tree re-organization steps during loading. While query time decreases as expected, given the fewer nodes and candidates, the most important time consideration is clearly the initial index construction. While this is essentially a pre-process step in our setup, it is worthwhile to note for real world application and index maintenance.

Conclusions and Future Work

We show that iDStar performs well in tightly-clustered and high-dimensional dataspaces. Unfortunately, the curse of dimensionality still remains an issue beyond 256 dimensions, where Euclidean distance offers little differentiation between data points (and index values). We find the number of partitions used can affect the quality of the index beyond the general performance metrics of candidates, nodes, and

[1] http://corpus-texmex.irisa.fr/

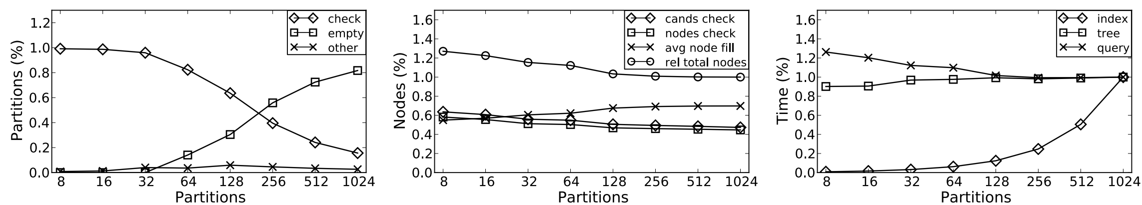

Figure 6: Varying the number of partitions used with iDistance on real data.

query time. The pattern of performance changes on real data supports prior findings with new insights into why iDistance performance plateaus around $2D$ partitions.

Future work continues the development of iDStar into a fully-fledged indexing algorithm, with specific focus on real-world applications, extended optimizations, and a public repository for the community. We are further exploring novel EM algorithms to perform clustering for the sake of indexing which will improve reference point quality, as well as developing an iDStar subspace segmentation extension to greatly enhance performance by retaining useful Euclidean distance information in extremely high-dimensional spaces.

Acknowledgment

This work was supported in part by two NASA Grant Awards: 1) No. NNX09AB03G, and 2) No. NNX11AM13A.

References

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

Bayer, R., and McCreight, E. M. 1972. Organization and maintenance of large ordered indices. *Acta Informatica* 1:173–189.

Beckmann, N.; Kriegel, H.-P.; Schneider, R.; and Seeger, B. 1990. The R*-tree: an efficient and robust access method for points and rectangles. In *Proc. of the ACM SIGMOD Inter. Conf. on Mgmt. of Data*, volume 19, 322–331. ACM.

Bellman, R. 1957. *Dynamic Programming*. Princeton University Press.

Berchtold, S.; Böhm, C.; and Kriegal, H.-P. 1998. The pyramid-technique: Towards breaking the curse of dimensionality. *SIGMOD Rec.* 27:142–153.

Doulkeridis, C.; Vlachou, A.; Kotidis, Y.; and Vazirgiannis, M. 2007. Peer-to-peer similarity search in metric spaces. In *Proc. of the 33rd VLDB Conf.*, 986–997.

Guttman, A. 1984. R-trees: a dynamic index structure for spatial searching. In *Proc. of the ACM SIGMOD Inter. Conf. on Mgmt. of Data*, volume 14, 47–57. ACM.

Ilarri, S.; Mena, E.; and Illarramendi, A. 2006. Location-dependent queries in mobile contexts: Distributed processing using mobile agents. *IEEE Trans. on Mobile Computing* 5(8):1029–1043.

Indyk, P., and Motwani, R. 1998. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proc. of the 30th ACM Sym. on Theory of Computing*, STOC'98, 604–613. ACM.

Jagadish, H. V.; Ooi, B. C.; Tan, K. L.; Yu, C.; and Zhang, R. 2005. iDistance: An adaptive B+-tree based indexing method for nearest neighbor search. *ACM Transactions on Database Systems* 30(2):364–397.

Lowe, D. 1999. Object recognition from local scale-invariant features. In *Proc. of the 7th IEEE Inter. Conf. on Computer Vision*, volume 2, 1150–1157.

Ooi, B. C.; Tan, K.-L.; Yu, C.; and Bressan, S. 2000. Indexing the edges: a simple and yet efficient approach to high-dimensional indexing. In *Proc. of the 19th ACM SIGMOD-SIGACT-SIGART Sym. on Principles of DB Systems*, 166–174. ACM.

Qu, L.; Chen, Y.; and Yang, X. 2008. iDistance based interactive visual surveillance retrieval algorithm. In *Intelligent Computation Technology and Automation (ICICTA)*, volume 1, 71–75. IEEE.

Schuh, M. A.; Wylie, T.; Banda, J. M.; and Angryk, R. A. 2013. A comprehensive study of idistance partitioning strategies for knn queries and high-dimensional data indexing. In *Big Data*, volume 7968. Springer. 238–252.

Schuh, M. A.; Wylie, T.; and Angryk, R. A. 2013. Improving the performance of high-dimensional knn retrieval through localized dataspace segmentation and hybrid indexing. In *Proc. of the 17th ADBIS Conf.*, 344–357. Springer.

Shen, H. T.; Ooi, B. C.; and Zhou, X. 2005. Towards effective indexing for very large video sequence database. In *Proc. of the ACM SIGMOD Inter. Conf. on Mgmt. of Data*, 730–741. ACM.

Singh, V., and Singh, A. K. 2012. Simp: accurate and efficient near neighbor search in high dimensional spaces. In *Proc. of the 15th Inter. Conf. on Extending Database Technology*, 492–503. ACM.

Tao, Y.; Yi, K.; Sheng, C.; and Kalnis, P. 2009. Quality and efficiency in high dimensional nearest neighbor search. In *Proc. of the ACM SIGMOD Inter. Conf. on Mgmt. of Data*, 563–576. ACM.

Wylie, T.; Schuh, M. A.; Sheppard, J.; and Angryk, R. A. 2013. Cluster analysis for optimal indexing. In *Proc. of the 26th FLAIRS Conf.* AAAI.

Yu, C.; Ooi, B. C.; Tan, K.-L.; and Jagadish, H. V. 2001. Indexing the Distance: An Efficient Method to KNN Processing. In *Proc. of the 27th VLDB Conf.*, 421–430.

Zhang, J.; Zhou, X.; Wang, W.; Shi, B.; and Pei, J. 2006. Using high dimensional indexes to support relevance feedback based interactive images retrieval. In *Proc. of the 32nd VLDB Conf.*, 1211–1214. VLDB Endowment.