SLASH - Single-Link Agglomerative Scalable Hilbert Clustering

Paul Anton Chernoch

Abstract

This paper introduces SLASH, an unassisted clustering algorithm for high dimensional data that scales linearly with the number of dimensions, both in time and in storage. It reports on experiments that verify its scalability and applicability to problems where the clusters can be spherical or oddly-shaped, may be of the same or widely varying sizes, and may be of uniform or different densities. SLASH is fairly insensitive to outliers and the original ordering of the points. It can tolerate a certain amount of noise and partial overlap of clusters.

SLASH derives its qualities from the way it exploits properties of the D-dimensional Hilbert Curve to overcome the "curse of dimensionality". It makes use of fast but greedy single-link agglomeration for its initial clustering, combines clusters further using other slower approaches, then refines its results using nonparametric density-based agglomeration to reprocess individual clusters and potentially divide them.

SLASH uses Euclidean distance (the L_2 norm) as its metric. It assigns each point a single classification. The resulting partitioning of clusters is flat, not arranged in a hierarchy.

SLASH is a useful tool in its own right, but pieces of it can be used to enhance existing clustering algorithms, such as helping K-MEANS decide on which cluster count K to use or supplying a density measure to a density-based clustering algorithm. The complete source code, written in C#, is available on Github.

Keywords: clustering, Hilbert curve, machine learning, classification, high dimensions, nonparametric density

Overview: The What, When and Why of Using SLASH

Unassisted clustering of data is a vital machine learning task with wide application to data mining, document retrieval, bioinformatics and other active areas of research. None of the myriad algorithms invented to tackle these problems possess all the desired characteristics or work with all kinds of data. This paper proposes an algorithm with the acronym SLASH that employs the Hilbert Curve to assist in clustering data. It identifies the types of problem best suited to using the algorithm, explains how its execution time and memory scale well with the problem size, and ponders future research that might improve and extend it.

Classification. Taxonomies of clustering algorithms have four broad categories: *partitioning* (like the popular K-MEANS method), *hierarchical* (which may be top-down, like CURE and ROCK, or bottom-up, like BIRCH), *density-based* (like DBSCAN), or *grid-based* (like CLIQUE). [1] However, hybrid algorithms like SLASH are best described by their attributes.

SLASH is *partitional* (flat), dividing points into clusters without regard to any hierarchy.

SLASH assigns every point to a *single category*, not multiple categories.

SLASH makes decisions using both *local metrics,* such as single-link distance and density, and *global* ones, such as minimizing the diameter of clusters being merged.

SLASH performs *bottom-up agglomeration*, not top-down division.

Applicability. When should SLASH be used? For smaller numbers of dimensions D, other algorithms shine. SLASH is best when 2^D exceeds the number of points N, typically 15 dimensions or higher. SLASH can handle clusters that vary widely in size and are oddly shaped, even have large differences in density. In this regard it is superior to K-MEANS, which is best

suited to spherical clusters of like size. [2] SLASH is easier to configure, because it includes a step that estimates the number of clusters K, whereas K-MEANS and related methods require that you supply K as an input parameter. [2] It also derives the linkage distance L and the neighborhood radius for the density calculations. SLASH employs techniques to ameliorate the effects of outliers and noise, although more work on the latter is needed. Euclidean distances are used, so SLASH is restricted to data best studied using that metric. (However, a technique for extending the usefulness of the Euclidean metric to binary attributes is presented.) Lastly, because of the Hilbert curve sorting, SLASH is indifferent to the original ordering of the points.

Time. The bottom line is performance. The running time varies with these quantities:

N – number of points
D – dimensions per point
K – clusters (excluding small outliers)
B – bits required to represent the largest coordinate among all points

Different phases of SLASH have different run time dependence:

1) Hilbert transform: O(N • D • B)
2) Sort points in Hilbert curve order: O(N • Log N • D • B)
3) Sort distances while finding linkage distance: O(N • Log K)
4) Rough single-link, agglomerative clustering : O(N • D • B)

The crucial result is that no phase has worse than a linear time-dependence on D!

Space. The good news continues when it comes to memory requirements. Some implementations of K-MEANS as well as some grid-based approaches use R*trees, Kd-trees or quadtrees and the like to speed up nearest neighbor searches. R*trees begin to suffer from problems of node overlap at five dimensions, leading to poor performance. [3] The other structures require that one have at least 2^D points in the dataset, otherwise most cells will be empty or have a single point in them, and the utility will vanish. [4] Similarly, some algorithms for the Hilbert transformation require state diagram structures whose size grows exponentially with D. [5] They cannot be used beyond about twelve dimensions. The Hilbert curve algorithm employed by SLASH has a space requirement linear in D.

Having asserted the characteristics of SLASH, next we explore how it achieves these results and overcomes the "curse of dimensionality" so aptly named by Richard E. Bellman in 1957.

The Ideal Clustering Algorithm (and SLASH)

The ideal clustering algorithm discovers – unassisted – patterns that were not visible before, or automates what was visible to humans but previously required time-intensive, manual classification. It does so with a minimum of computer and human resources, and it scales well to accommodate the ever-larger "big data" problems arriving on our doorstep. Here is how SLASH stacks up against various issues vital to clustering.

"**N**". The ideal algorithm would grow no faster than N, the number of data points. Given that the execution time for some clustering algorithms is quadratic or even cubic in N, this is a challenge. [14] Most parts of SLASH are linear, but two sort operations are N Log N. One can be reduced to N Log K if one estimates and imposes an upper bound on K, or N Log $\sqrt{N}$ if one concedes that there are likely fewer clusters than $\sqrt{N}$. However, sorting according to the Hilbert curve position cannot be done in less than N Log N time. Nevertheless, while this suffers in comparison to some other methods in its dependence on N, it quickly makes up for it as D increases, both in time and in storage.

The main clustering step, the single-link agglomeration, has a time dependency proportional to N. This is possible because a single scan of the points in Hilbert curve order finds enough linkages between adjacent points worthy of merging to accomplish the majority of the clustering.

"**D**". Ideally, the method would be no worse than linear in D, the number of dimensions. The sensitivity to D pops up in surprising ways due to the "curse of dimensionality". The first problem is loss of contrast. [6] Consider a random sample of pairs of points drawn from the data. As the number of dimensions increases, the ratio in distance between the closest pair and the farthest pair tends toward unity. All points start to look as though they are equidistant from all other points. In any linkage-based agglomerative method (such as single-link, complete-link or average link), key decisions about whether to merge points into the same cluster or not are based on the Euclidean distance between points. As contrast diminishes, this approach suffers.

The solution provided by SLASH is to use the Hilbert curve to sharpen the contrast so that two things may be found efficiently and accurately:

1. The linkage distance L
2. Many pairs of points that are separated by no more than L

In an experiment using 10,000 points in fifty dimensions, 10,000 random pairs of points were selected. Over 98% of the pairs were separated by distances between 2,000 and 3,500, a contrast of 1:1.75. That distance greatly exceeded the size of the gaps between clusters. This means that very few pairs of points worthy of being merged were found. Only 1% of the pairs' distances were below 250, a value close to the optimal linkage distance. As the number of dimensions increases, that contrast will worsen, and the number of near neighbors found by random search will drop to zero.

On the other hand, when taking pairs of consecutive points sorted according to the Hilbert curve, over 98% of the pairs were separated by a distance of 282 or less. By collecting those distances, sorting them, and identifying where the distance series makes a sharp increase, one can find the linkage distance L to use. Then if you merge all pairs of consecutive points whose distance does not exceed L, you arrive at a rough first clustering, having analyzed only N-1 pairs of points! (The Hilbert curve was chosen over other space filling curves because it fragments data less than others, such as Z-order, raster and Peano. [7])

The clustering approach just described handles a second problem related to high dimensional spaces: the k-nearest neighbor problem. The Hilbert curve does not give you all of a point's nearest neighbors, but it gives you many of them. Follow-up steps will help find more near neighbors worth merging and reduce the number of clusters further.

The next problem with high dimensions relates to finding the density in the vicinity of a point. [8] Consider two ways of counting neighbors: bounding hypercubes and bounding hyper-spheres. The first lends itself to a grid structure. However, as noted, as the number of dimensions increases, the number of grid cells increases at 2^D. That is not the worst consequence. Consider the ratio of the area of a circle to its enclosing square: π/4. In three dimensions the ratio of the volume of a sphere to its enclosing cube drops to π/6. As the dimensionality increases, this ratio rapidly converges towards zero: [9]

$$\lim_{d \to \infty} \frac{\pi^{d/2}}{d2^{d-1}\Gamma(d/2)} = 0$$

Hypercubes are likely to vastly over-estimate the density. Moreover, to use either approach, you need to decide how large a volume to use for the "neighborhood" of your density calculations. Too small a volume and most points will have zero neighbors. Too large a volume and you lose the capacity to discriminate, as larger volumes suck up similar numbers of points. Then you need to find and count all the neighbors in that volume, a potentially $O(N^2)$ task.

One might reason that if you find among the neighbors of point A a second point B, that A and B must share many near neighbors in common, so finding B's neighbors would take less time, etc. Sadly, in high dimensions, this is false. The set of neighbors of neighbors of A seldom includes A. You cannot exploit transitivity to improve efficiency. [10]

Happily, the Hilbert curve can assist. The linkage distance (L) found earlier is already known to link most points to a neighbor on each side along the curve. If we use it to define our neighborhood radius, we will not have the problem of zero density. However, in practice, this distance is too high, producing too many neighbors and requiring too large a window along the Hilbert curve for our search. Experiment shows that L/3 is too small, producing too many zero density measurements. L/2 is close. We chose 0.4L. Then, taking a window of points to the left and right of the test point along the Hilbert curve, we compare the test point to its neighbors and count how many fall inside the neighborhood radius. The size of this window along the Hilbert curve is crucial. Too small and we either won't find enough neighbors or will saturate. Saturation occurs when the actual number of neighbors is larger than the window size, causing the count to be truncated. Either will cause the value to diverge from the true value. Too large a window and the execution time suffers. Experiment shows that a window radius of $\sqrt{(N/2)}$ yields an acceptable estimate of the neighbor count as compared to an exhaustive search. For our purposes, we only require that this estimate be highly correlated with the true value. Using 0.4L as the neighborhood radius (in D-dimensional space) and $\sqrt{(N/2)}$ as the window radius (along the 1-D Hilbert curve) generally yields a worst case value of 0.75 for Kendall Tau-B correlation, and often over 0.8. That correlation is adequate for use in the density-based agglomerative clustering step that deals with discriminating improperly merged clusters that partially overlap, or have noise points between them forming a data isthmus, which often fools single-link algorithms into merging clusters that belong apart.

"**K**". The number of clusters K has no impact on the performance of some clustering algorithms but significant effects on others. [11] SLASH follows its initial clustering pass with a refinement pass that compares each cluster to a certain number of nearby clusters to see if their centroids are close enough to merge. This step can be proportional to either K or K^2, depending on configuration. To do this we must solve the polychromatic closest pair problem (PCCP) and merge clusters that contain points that are close to one another. [12] The algorithm used for PCCP has running time proportional to $(K^2) \cdot (2N/K) = 2KN$, where N/K is the average size of a cluster.

K-MEANS is sensitive to the initial choice of K, but not SLASH, so long as the estimate is on the high side, which it is in practice. SLASH estimates K and this estimate drives the speed of the second phase, not its accuracy. The initial estimate has been observed experimentally to fall in the range 1.3 to 3 times the true K, regardless of problem size. An algorithm called ClusterCounter performs a downhill search for the best Hilbert index. The better the index it finds, the lower the estimate for K. This is possible because there are D! (factorial) possible Hilbert curves that can be generated for points of dimension D. Each curve is a complex rotation and twisting of the original curve, and they pass through the data points in a different order. Each curve is formed by generating a random permutation of the coordinates of the data and applying that permutation to all data points. By trying many such permutations and gradually reducing the number of coordinates scrambled at each stage, we settle on a local minima. The

selected curve will yield the lowest estimate of K that our time budget permits. This optimization phase usually succeeds in reducing K to twice the true cluster count or less.

Geometrically, this is what happens. As you follow the points along the Hilbert curve, you enter one cluster, visit multiple points from that cluster, leave it for another cluster, wind through that group, revisit the first cluster for more of its points, and continue visiting other clusters. Some clusters you visit only once, some twice, some more often. An ordering is fully concordant with the clustering if you visit each cluster only once. If each cluster is visited exactly once, the curve concordance is perfect and the estimated K equals the actual K. Since each Hilbert curve is different, some are bound to be more concordant than others. The more concordant a curve we find, the more efficient will be the subsequent steps in the algorithm.

"**B**". The number of bits per dimension corresponds to the granularity of the data, how many bits are needed to encode the largest value taken from all the coordinates of all the points. Most clustering algorithms use standard integers or doubles and pay no attention to the possibility of shrinking the representation to a smaller word size. The Hilbert transformation, however, is dependent upon this value and scales linearly with it.

Noise. Noise complicates any algorithm. One way it adversely affects SLASH is in the selection of the linkage distance L. All the pairwise distances are measured for consecutive points taken in Hilbert curve order, then sorted from low to high. If points within clusters are close and clusters are well separated, a histogram of the observed distances between points will show a clear gap, a range of distances that separates comparatively few pairs of points. This shows up by studying the ratio of the increase in distances from one pair to the next. When that ratio jumps the most, we have moved from the regime of distances between neighbors within a cluster to distances between clusters. However, if you add noise points, they will blur the distinction between intra- and inter-cluster distances. To compensate, SLASH compares the relative increase in distances not between adjacent pairs but between pairs separated by a gap called NoiseSkipBy, which will exaggerate the effect. An unfortunate side effect of this is that the length of the gap is added to the estimated K.

A second way that noise (or outliers) intrudes is by forming bridges between two clusters, causing them to be incorrectly merged. One way to compensate for this is to perform a series of clusterings of random subsets of the points, then blend the results by voting on whether certain points should remain grouped together. If the fraction of points chosen is small enough, the odds favor the bridge being broken if it is narrow enough. The primary way that SLASH handles noise is by permitting the clusters to be merged at first, only to break them apart again later using density-based agglomeration. Experiments show that Gaussian clusters with 65% overlap can usually be separated using this approach.

Density variations. Many clustering algorithms suffer when some clusters are very dense while others are diffuse. [10] So long as the average spacing inside the diffuse clusters is still much smaller than the smallest inter-cluster distance, SLASH can accommodate such variation without difficulty.

Online vs Offline. Adding new observations into an already clustered universe and shunting them to existing clusters or re-computing the clusters is a desirable trait. SLASH has not been designed to perform online clustering. However, adding new points to a Hilbert index, quickly finding their nearest neighbors along the curve and inserting them into the existing categories would be a simple modification. Inserting into the index can be done using a binary search, which is O(Log N), followed by an insertion, which is O(N) for a List. Then the search for the closest cluster would be a constant time operation, so long as its neighbor on the Hilbert curve is within a distance of L.

Uneven Size. K-MEANS tends to form clusters of equal size. [2] SLASH has no difficulty handling clusters that vary in size. However, you must specify an "outlier size". All clusters that are smaller than this will be joined to the nearest larger cluster during the outlier phase.

Irregular Shape. K-MEANS tends to generate spherical clusters. [2] SLASH can handle any shape so long as the different regions are reachable via links of length L. However, if the cluster has several distinct density centers, the density-based phase might divide it into pieces.

As a caveat, the secondary clustering which joins large clusters to other large clusters can be confused by irregularly shaped clusters. The approximate PCCP algorithm may overestimate the cluster distance. Using the exact PCCP algorithm that is employed in the Outlier Clustering phase would fix this at the expense of much worse execution time: $O(N^2)$ versus $O(3N)$.

Point ordering. Some algorithms are sensitive to the order in which points are processed and can yield different results. SLASH always orders the points according to the Hilbert curve, hence yields consistent results.

Contrast. As discussed earlier, the poor contrast of high dimensional data is sharpened in SLASH by using the Hilbert curve.

Flat vs Hierarchical. SLASH is designed to be flat. However, one could alter the algorithm to build a hierarchy. One level would result from the initial single-link clustering. The second level would come from joining large clusters to other large clusters. A third level would come from joining outliers to the nearest clusters, Then when large clusters are divided via the density-based agglomeration phase, the resulting pieces would form another level.

Single vs multiple classification. SLASH confers a single classification to points. To generate multiple classifications, the clustering should be run multiple times against sub-spaces within the data, i.e. subsets of dimensions.

Local vs global considerations. SLASH employs two local measures to decide which points to merge, single-link distances and local point densities. However, the use of the Hilbert curve to select the linkage distance and order the densities injects a degree of global perspective to the procedure. Additionally, a final recombination step decides by minimizing the radius of the combined versus the distinct clusters, a global measure.

Single-link vs Complete-link vs Average-link. With single-link agglomeration, a few points in two clusters close together can cause a merge. If using Complete-link agglomeration, two clusters separated by a little bit more distance but with many points of close contact would be preferred instead. [14] The density-based agglomeration phase compensates for this.

Density vs Linkage vs Graph-reachability. SLASH benefits from the complementary use of density-based and linkage-based clustering, enjoying the advantages of each. Even better results can be gained using graph reachability clustering, but the time and memory requirements for building and traversing the graphs are considerable. Graph reachability is particularly good at identifying and severing data isthmuses. [10]

Ease of Configuration. Finally, the best clustering algorithms force the user to set few parameters before using them. K-MEANS requires that you supply K and choose an initial set of cluster centers. Poor choices by the caller lead to poor clustering and slow convergence, respectively. SLASH has its share of parameters, but research is showing that if most are set to default values the clustering that results is good across a wide range of problems. Among the parameters are:

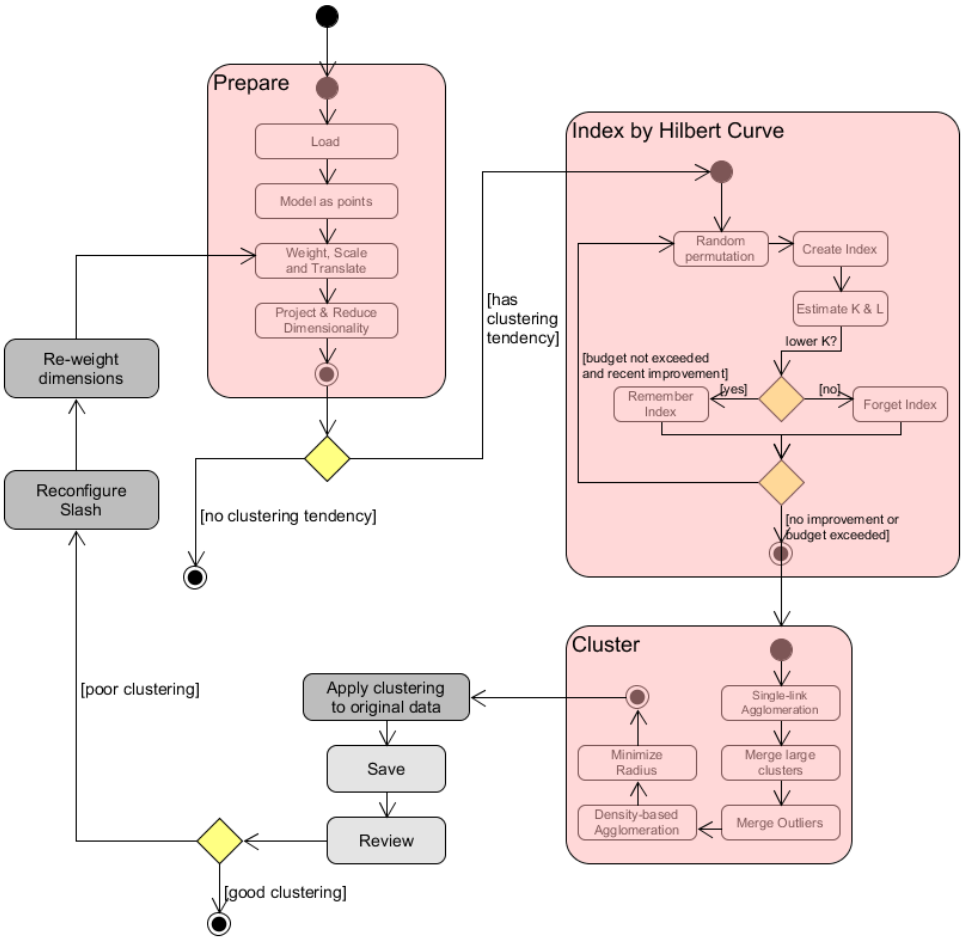
- optimization budget and convergence criteria for finding the best Hilbert index
- NoiseSkipBy, needed to compensate for noise when computing the linkage distance L needed for the first round of clustering
- # of neighboring large clusters to compare to each large cluster during the second round
- outlier size, which influences the second and third rounds

- window radius, which influences the density-based fourth round
- unmergeable size, which defines the threshold at which density centers are prevented from merging with other density centers, also in the fourth round
- radius shrinkage

All the above have acceptable defaults, one of which (window radius) grows as √N.

SLASH Algorithm Details

Here is an outline of the algorithm, followed by comments on steps that require elucidation.



UML Activity Diagram of Slash

A. Prepare Data
    1. Select Relevant Attributes
    2. Model Attributes as Numbers
    3. Scale, Translate and Quantize Attributes
    4. Remove Redundant Dimensions
    5. Assess clustering tendency
B. Search for an Optimal Hilbert Index
    1. Loop until Budget Exceeded or No Improvement in K
    2. Generate Random Permutation of Dimensions
    3. Apply Hilbert Transformation to Points
    4. Sort Points by Hilbert Index
    5. Measure Distance Between Consecutive Pairs of Sorted Points
    6. Sort Distances and Find Where they Increase Fastest
    7. Compute Linkage Distance L
    8. Estimate Cluster Count K
    9. Select Index with Lowest K
C. Primary Clustering via Single-link Agglomeration
    1. Start with N Points in N Separate Clusters
    2. Loop over Hilbert Index Sorted Points
    3. Measure Distance d between Every Consecutive Pair of Points
    4. If d ≤ L, Merge Points into Same Cluster
D. Secondary Clustering of Large Groups using Centroids
    1. Compute Cluster Centroids where Cluster Size S ≥ OutlierSize
    2. Loop over Clusters where S ≥ OutlierSize
    3. Compute Distance to all other Large Cluster Centroids
    4. Sort by Ascending Distance between Centroids
    5. Pick a Configured Number of the Nearest Neighboring Clusters
    6. Estimate the Polychromatic Closest Pairs of Points between Clusters
    7. If Distance between Closest Pair of Points ≤ L, Merge Clusters
E. Tertiary Clustering of Small Outliers
    Loop through Outlier Clusters with size S < OutlierSize
        Loop through Large Clusters with size S ≥ OutlierSize
            Compare Centroids of each Outlier with each Large Cluster and Sort
            Loop over Several of the Nearer of the Neighboring Clusters
                Exhaustive PCCP between Each Pairing of a Large Cluster with an Outlier
                Sort Ascending by Distance d, take Nearest only
                If d ≤ configured multiple of L
                    Merge the Outlier with its true Nearest Neighboring Large Cluster
F. Final Density-based Reclustering of Selected Clusters
    Loop over Clusters Identified as Potentially Requiring a Split
        Start a new Classification containing just the Points from the original Cluster
        Put Each Point in its own Cluster
        Create a new Hilbert Index for the subset
        Derive Neighborhood Radius from L
        Derive Window Size from N.
        Loop over all Points and Estimate Density given N and L
        Sort all Consecutive Pairs of Points Adjacent along the Hilbert Curve by Density
        Loop over Pairs from High Density to Low
            If Distance ≤ L, Potentially Merge the Points into the same Cluster
                If both Clusters to be Merged have S ≥ Unmergeable Size, Prevent Merge
                Otherwise, perform Merge
        Merge Outliers
        Merge to Minimize Radius
        Fold the Reclustered Points back into the main Classification
G. Apply Clustering to original data
H. Save results

Preparing Data. There is an art to modeling entities as numerical points in a multidimensional space. Here are examples from a project to identify overseas bus tours that were similar enough to be consolidated. One challenge was to model attributes that were non-numerical. One attribute was called "tour direction". Some tour groups preferred to visit the cities in the order given in the brochure, which was called "forward". Some groups preferred to travel in "reverse". Some groups had no preference. A workable solution was to model "forward" as 10, "indifferent" as 20, and "reverse" as 30. This way "forward" is far from "reverse" but near "indifferent". A single dimension can be used to encode all three.

If all choices for an enumerated attribute should be considered equidistant from one another, they should each get the same coordinate value, but be stored in separate dimensions. To handle the tour itinerary, a separate dimension was created for each city that at least one tour visited, some 160 in all. If a tour visited a given city, that dimension would receive a value, otherwise zero. (This is similar to the binary attributes used when clustering documents modeled as a "bag of words". For such applications, the cosine or jaccard similarity measures are preferred by many. However, there is a way to model binary attributes for use with a Euclidean metric that addresses the difficult cases raised by Ertöz and others. [10] This modeling approach will be discussed later, alongside the results of the experiments.)

The desired departure date dimension was easiest, with January 1 assigned the value 1, January 2 the value 2, all the way to December 31 as 365 (or 366).

Once all attributes have been modeled as numbers, they must be scaled. Some attributes are more important than others. We could decide to modify the metric function to weight different coordinates unevenly, but decided instead to scale the data first, then use the straightforward Euclidean distance formula. The scaling is best done by performing many nearest neighbor searches and manually inspecting the results. If an unimportant similarity causes two points to be viewed as neighbors, rescaling the dimensions can fix that. In the tourism problem, cities and departure date proved critical, while most other attributes were secondary.

With the data modeled and scaled, it still must be made suitable for the Hilbert transform. All coordinates must be non-negative integers, which may require translation of negative values and scaling of fractions to make them whole. You must decide how coarsely to quantize continuous variables. The log base 2 of the largest coordinate value rounded up is the number of bits B required per coordinate. This number affects algorithm speed and memory usage.

Next, one may wish to perform PCA (Principal Component Analysis) [15], Projection Pursuit [16], or some other technique for finding redundant dimensions and removing them. For example, if one dimension is a linear combination of two other dimensions, it can be eliminated. The fewer the dimensions, the faster the computation, less memory used, and less muddy the results.

Finally, discard duplicates. If after the previous transformations, several points are duplicates, toss all but one and group the like points together. After clustering of the unique points is complete, the duplicates can be folded in. This reduction can be significant. In one problem tackled by the author involving optimizing delivery times from every supplier ZIP code to every possible customer ZIP code, the number of points was reduced from over 40,000 to about 960.

Test Clustering Tendency. If the data doesn't exhibit any clustering tendency, it is pointless to perform the many computations that follow. There are many ways to do this, but since we will be discussing the uses of the Hilbert curve, here is a means that employs the curve to quickly assess clustering tendency:

1. Compute the largest value among all points and dimensions, then find M, the smallest power of two greater than this value.
2. Translate the coordinates of all points so that the median value for each coordinate is shifted to coincide with M/2. This will as closely as possible divide all points into two equal halves at the midpoint of each coordinate's range.
3. Compute a coarse Hilbert transform using a single bit for each coordinate.
4. Compare the number of distinct Hilbert Index values to the number of points. If the numbers are nearly equal, there is no clustering tendency.
5. Also establish what proportion of points have the most common index value. If almost all points have either this common index value or are by themselves or in groups smaller than the outlier size, then we likely have one big cluster and lots of noise.

When this test was applied to uniform fifty-dimensional random data (i.e. unclustered points), every point yielded a different index. When applied to well-separated Gaussian clusters, most points were in large groupings, and the number of such groupings was not much different from the actual number of clusters. Because only one bit per dimensions was used, this test is independent of the number of bits needed to represent the data. Furthermore, because a dictionary was used to aggregate, no sorting was required, hence the procedure is O(N•D).

Searching for a Hilbert Index. The hardest thing about working with the Hilbert curve is performing the transformation in a scalable manner. After months of failed attempts at writing such an algorithm or finding a fast one in the literature, one turned up. [17] The code is linear in dimensions D and bits B, both in time and in memory. (The published code has a typo, rendering it unusable. This typo is cited and corrected in the SLASH source code.)

The next question that arose was that since many different Hilbert curves can be generated for a set of points, are some better than others for clustering? If so, how would you compare them? Next, is there a method to find a good one if you don't already know how the data is clustered?

The first approach began by designing a measure called Concordance. Given a curve and a gold standard for the clustering, walk the curve in Hilbert Index order. Every time we enter a new cluster, add one. A perfectly concordant curve would enter each cluster once, visit all its points and miss none, then advance to the next cluster. The worst curve would skip to a new cluster at each point, for N visits in all:

$$C = \frac{K}{V}$$

The value of C can range from 1 for perfection to K/N for the worst case. This external metric proved useful for verifying that the more concordant a curve, the better a job it did while clustering. What about an internal metric that works when you don't know the answer? The insight here was that points within a cluster are near one another, while points in different clusters are farther apart. By summing up the distances between consecutive points along the

curve we can derive a path length. Perhaps curves with a shorter path length were more concordant? Experiments showed that this was true.

However, the correlation between the path length and concordance was not perfect. The next experiment was to use the full cluster counter to estimate K and discount any outliers. This proved more successful. Why? If the curve takes an excursion from one outlier to the next in a random order, it can rack up a lot of distance without affecting the cluster count. Thus the best measure of a good curve is the one which yields the fewest large clusters.

With an internal way to evaluate quality, it became possible to create a downhill optimizer to search for quality curves. We begin by randomly scrambling all coordinates, try several curves, then keep the best. At each stage we randomly scramble a smaller number of coordinates, until we make no improvement. This approach often cuts the initial estimate of K in half, making it much more accurate and promoting efficiency in later steps that depend on this initial K value.

Creating so many Hilbert indices is costly, especially in memory, as they are created in parallel. One way we reduce the cost is to evaluate permutations using a sample of points, not all N. However, if our sample is too small, it will suffer in its ability to resolve all the clusters and may undercount K, which is bad. Thus the first count uses all the points. Given this initial K, we decide how small the sample size can be yet not cause any clusters to shrink to the size of an outlier. Once we have optimized K for the sample, we run it once more on the full set of points.

The final word on the index is how it estimates the linkage distance L and cluster count K. Having gathered the pairwise segment distances between points along the curve and sorted them, imagine we saw this:

...230, 245, 259, 280, **300**, **1200**, 1800, 2200, 2700...

Note the interval from 300 to 1200. The difference of 900 is the largest shown, and the ratio of 4 is the largest as well. When an interval has both the largest difference and ratio, the lower number of the pair is chosen as the linkage distance L. If one interval has the largest increase but a different interval has the largest ratio, we prefer the maximum increase if the distance is below the mean, or the maximum ratio otherwise. The reason is that an extreme outlier is likely to be a large distance from its neighbor, but not necessarily by a large multiplier over the next observed distance. Conversely, early on, if one point has d = 1 and the next has d = 10, the multiplier is misleadingly high. We ignore multipliers before d = 10. (There are more tests to deal with other edge cases.)

With L in hand, we loop over the points in Hilbert order and count how many times the distance exceeds L, but exclude instances where the cluster we are leaving is an outlier. This count plus one is K, the estimated cluster count.

Primary Clustering via Single-link Agglomeration. Before describing how the Hilbert curve is useful in single-link agglomeration, we will step back and look at two basic algorithms. The first is a naive hierarchical algorithm:

1. Compute all N^2 point-pair distances.
2. Sort pairs by Ascending distance.
3. Loop over all point-pairs in sorted order (closest pairs first) and combine into clusters to form a hierarchy.

This algorithm requires $O(N^2 \text{ Log } N)$ time, and gives no insight into when we have moved from tight intra-cluster distances to the merging of distinct adjacent clusters. Next is a flat partitioning algorithm:

1. Compute all N² point-pair distances.
2. Identify a characteristic link distance L.
3. Loop over all point-pairs in any order and merge pairs with distance d ≤ L.

Apart from step 2, the flat approach only requires O(N²) time, because it does not have to perform a sort. However, the discovery of L is complicated by density variations, and may require sorting the distances, yielding time again proportional to O(N² Log N).

SLASH differs from these two approaches because it will use the Hilbert curve to:

1. Compute only O(N) point-pair distances, drawn from consecutive points along the curve.
2. Sort the pair distances, for O(N Log N) time.
3. Loop over sorted point-pairs to discover L, for O(N) time.
4. Loop over point-pairs again (in any order), merging pairs where d ≤ L, for O(N) time.

To elaborate, if consecutive pairs of points taken from along the Hilbert curve are separated by d ≤ L, they are merged together. When a point that has already been merged with other points is merged again, all the points grouped with either of the points being merged end up in a single cluster.

One way to improve this phase is to use a frugal streaming algorithm to estimate distance percentiles in a single pass, thus avoiding the sorting in step 2. [21] This may introduce additional imprecision to the calculation of L and K, the cluster count estimate. A second way to improve this phase (perform more merges) is by using more Hilbert curves, however there is a point of diminishing returns. More research is needed in this area.

Now is a fitting time to discuss the essential calculation that undergirds every piece of this algorithm: the Euclidean point-to-point distance formula.

$$d = \sum_{i=1}^{D} (x_i - y_i)^2$$

We use the square distance to avoid costly square roots. In addition, the code uses loop-unrolling, taking four terms of the series at a time. These are common optimizations, but we found more. We can distribute and regroup to pull the magnitudes of the vectors out as invariants:

$$d = \sum_{i=1}^{D} x_i^2 + \sum_{i=1}^{D} y_i^2 - 2 \sum_{i=1}^{D} x_i y_i$$

If the magnitudes (the first two summations) are precomputed and stored with each point, a third of the operations can be eliminated. This is the extent of the optimization of distance, but sometimes one does not need to know the distance, just whether it is larger or smaller than a given value. The classification method often tests to see if d ≤ L. Imagine two points in one dimension. Both can be on the same side of the origin or on opposite sides of the origin. If we know the magnitude of their respective distances from the origin, we can form an inequality:

$$|A - B| \leq d \leq A + B$$

If we shift to multiple dimensions and restrict values to the positive quadrant, then the farthest apart positions to place two points with given magnitudes is on different axes. The nearest they can be is collinear with the origin. This yields the following inequality:

$$|A - B| \leq d \leq \sqrt{\left(A^2 + B^2\right)}$$

If the distance to be tested is below the difference in magnitudes (left-hand side) or above the maximum distance, we can evaluate the inequality without computing the full distance. If used with arbitrary distances, we very seldom can exploit this optimization. However, if tested against linkage distance L, the optimization is possible in from 20-40% of cases. If one adds a second triangulation point at the opposite corner, where all coordinate values are at the maximum, the odds improve, and 26-47% of cases can benefit from the optimization. Additional research into the careful choice of a few other triangulation points may be a cost effective way to improve that percentage further. The benefit is that for those tests that do not need the full computation of the distance, the question is answered in constant time independent of dimensions D.

Secondary Clustering of Large Groups using Centroids. The goal of this phase of clustering is to combine neighboring large clusters overlooked by the previous phase because the Hilbert curve followed a path that fragmented the clusters. It compares clusters formed in the previous phase that are not outliers (smaller than the outlier size) to other such clusters. If the clusters are close enough, merge them. The initial test of distance compares the centroids to eliminate bad candidates for the second, more expensive test. The second test solves the polychromatic closest pairs problem (PCCP) approximately. This approximation is good for spherical clusters, but not for oddly-shaped ones. The steps:

1. Find the centroid of the first cluster, C1.
2. Compare every point in the second cluster to find the point P2 closest to C1.
3. Compare every point in the first cluster to P2 to find the corresponding closest point P1 in the first cluster.
4. Assume P1 and P2 are the two closest points. Their distance d is taken as the distance between the clusters.
5. If d ≤ L, merge the clusters.

The algorithm is fast – O(N) – but a more accurate one for use with irregular clusters is desirable.

Tertiary Clustering of Small Outliers. Users of SLASH may prefer that truly distant outliers be left ungrouped. In place of infinity, one can configure SLASH to only merge outliers to the nearest large cluster if a threshold distance is not crossed.

Because outlier clusters are so small, often only one or two points, an exhaustive, accurate PCCP algorithm is used, since it will not suffer from $O(N^2)$ performance.

Final Density-based Reclustering of Selected Clusters. The idea behind the density-based algorithm is to perform single-link agglomeration as before, but to sort the merge

operations by the local density, from densest to most diffuse. This will cause seeds to form where the data is densest. Those seeds will grow. When a cluster reaches a certain "unmergeable" size, it will only be permitted to soak up individual points, outliers and small clusters. If everything works out, one density center will form for each cluster and grow until all nearby points are absorbed. The separate clusters will be prevented from merging, even if a path of short links with $d \leq L$ can connect them.

In practice, this works well, but sometimes a cluster will be split in more pieces than desired. A follow up step looks for "unmergeable" clusters that should in fact be merged. It measures the radius of each cluster, then computes the radius that a merger of those two clusters would have. If the sum of the radii of the two smaller clusters is more than the radius of the combined cluster, the clusters might be merged. The ratio of the two radius measurements we call the "shrinkage". If the shrinkage is less than a certain value (meaning a greater reduction in radius), the merge is done. Further research is needed to determine the best shrinkage ratio cutoff. Currently we use the value 0.6. The ideal value falls between 0.5 and 0.6. [18]

The density estimation is interesting in its own right. The density near a given point is derived by counting how many points in a window near the point along the Hilbert curve are within the neighborhood radius. The only cleverness involved is choosing the neighborhood radius and the window size. The first is taken to be a fraction of L (0.4L), while the second is taken as $\sqrt{(N/2)}$. The performance is tolerable because the N here is not that of the whole population, but merely the size of the cluster being split. This density estimator is much simpler than those described in the literature.

One key decision that SLASH does not yet know how to make is triaging which clusters to attempt to split using the density-based clusterer. [19]

Apply Clustering to original data. Care must be taken to match the ids in the original records to the points used to model them so that the final classification may be applied.

This completes the discussion of the algorithm. What follows are the results of experiments that demonstrate the claims made about its accuracy and performance.

Accuracy Experiments

Being close counts in horse races and clustering. It is often adequate if *most* points are categorized correctly. There are many formal measures of how close one classification is to another, such as Precision, Recall, F-measure, Adjusted Rand Index, Edit Distance and Entropy. [20] In testing SLASH, the BCubed measure of similarity was used. It blends together a measure of homogeneity with one of completeness. The first checks how often items that do not belong together are in the same cluster. The second checks how often items that belong together are in separate clusters. As a single value ranging from zero to one, it can tell you *if* something went wrong. As two measures, it can help explain *how* the algorithm went wrong. If homogeneity is high but completeness is low, individual large clusters have been incorrectly fragmented into many smaller ones. If the reverse is true, then clusters that should have remained separate were incorrectly combined. Of the two, the fragmented case is easier to fix.

The BCubed measure does not sample; it studies all points. This makes it an expensive computation. As an optimization, an order-independent hash of the point ids is generated for every cluster. If the set of hashes of the expected classification matches the hashes for the actual results, it is safe to assume BCubed is one (perfect).

Suitability of the Euclidean Distance Metric

Characteristics of the data or the goals of the analysis may dictate use of a metric besides Euclidean distance. High dimensional space can render it useless because of effects of the "curse of dimensionality". One problem case was clearly described by Ertöz et al and mimicked below. [10] Consider documents modeled as a "bag of words". Each document is abstracted as a vector of ones and zeroes, with each dimension corresponding to a keyword. A one means the document contains the word, a zero, that it is absent. Imagine four documents:

Document A: [1 0 0 0 0 0 0 0]
Document B: [0 0 0 0 0 0 0 1]
Document C: [0 1 1 1 1 1 1 1]
Document D: [1 1 1 1 1 1 1 0]

If you take the Cartesian distance between A and B, you get $\sqrt{2}$. If you do the same for C and D, you get the same distance! The first pair have nothing in common except that both lack most of the keywords. They are not very similar, they are merely not very dissimilar. The second pair of documents are almost identical and share all but two of the keywords. That these two pairs of documents should be considered the same distance apart makes no sense, hence using the Euclidean distance on such cases would compromise any clustering algorithm.

Serendipitously, there is a way to address this problem through a clever data modeling technique that exploits – rather than suffers from – the curse. Ideally, had we a large set of documents with no words in common, we would like to construct points equidistant from one another with a distance greater than zero, indicating they are not similar. On the plane, we can construct an equilateral triangle, but cannot place a fourth point such that all are equidistant from one another. In three dimensions, we can construct a pyramid with four equidistant vertices. For D dimensions, the most perfectly equidistant points we can construct is D+1. However, when we enter the realm of high dimensional space, if we choose points at random, the curse dictates that every point will be *nearly* equidistant from every other point. How "nearly" equidistant? In one test, points having one thousand dimensions were made by setting each coordinate to zero or one randomly. Here is a sample distribution of pairwise distances from a test run, comparing how far each of 10,000 points is from all the others:

Distance	# of point pairs
19	0
20	15
21	443,966
22	32,527,640
23	17,000,306
24	23,073
25	0

Since there are one thousand dimensions, on average half of the attribute values should match and half differ by one. The expected distance is therefore $\sqrt{500} \simeq 22.4$, which is consistent with the observed peak falling between 22 and 23. The spread in values is very tight.

By itself, this merely spreads missing attributes apart. We still need to deal with attributes that are set. The best arrangement is to assign a one to "word present" and randomly assign zero or two to "word absent". The symmetry means that a document that has a given word is equally distant from documents randomly given the value zero or two. Using this scheme, two documents having most words present will be very close, while documents having most words absent will be very far apart, thus circumventing the difficulty cited in the paper.

To test out this scheme, a random set of synthetic documents was generated having varying numbers of attributes, ranging from no words in common to all words in common. Four metrics were used to sort the distances between pairs of points: two Euclidean metrics (one using the traditional model and the other a randomized model), Jaccard distance, and cosine similarity. The first two were applied to the traditionally modeled points and the randomly modeled points, respectively, while the latter two were applied only to the traditional model of ones and zeroes with no randomly assigned values. Then Kendall Tau-B correlation (which permits ties) of the ranked sequences was performed. The correlation between the Jaccard measure and the old Euclidean method was poor, ranging from 0.25 to 0.45, while the correlation with the specially modeled points typically ranged from 0.45 to 0.6 (and occasionally as high as 0.8). The new method had a similar correlation with the cosine similarity (with opposite sign, as the latter is a similarity measure, not a distance measure).

The improvement (as compared to the traditional model) in correlation with two measures commonly used in comparing documents means that this approach should yield improved clustering results against documents modeled as collections of binary attributes. However, the correlation does not tell the whole story. Consider two documents that each contain eight keywords and share four in common, giving twelve distinct attributes in total. Their Jaccard similarity would be:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{4}{12} = \frac{1}{3}$$

Subtract this from one to get two-thirds, the Jaccard distance. Now repeat this for two documents that each contain sixteen keywords with eight in common, for twenty-four distinct keywords between them. The Jaccard distance is the same! Not so the Euclidean distance. With more points of similarity, the second pair of documents would be closer. Thus for such cases, this novel way of modeling binary attributes is an improvement.

Sharpening Contrast of Linkage Distance

The graph in figure 1 shows how 10,000 pairs of points chosen randomly from a population of 10,000 are 98+% likely to be far apart, while points adjacent along the Hilbert curve are 98% likely to be close enough to be clustered together. As the population size grows, this discrepancy will widen further. This demonstrates how effective the Hilbert Curve is at sharpening the contrast in distance between near and far points.

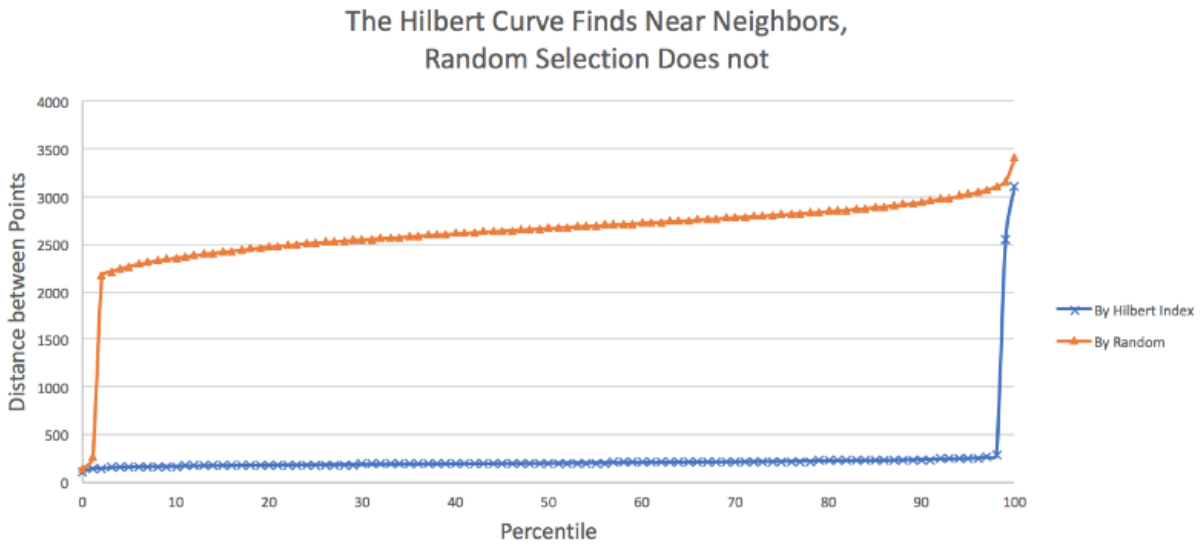


Figure 1. Contrast enhancing quality of the Hilbert Curve

Density

The density-based clustering phase relies upon the estimate of density correlating well with the actual density. This graph shows a typical example. The upper left edge is a sharp line with a slope of one. Those points are ones where the estimate and actual values match. At some points the estimate is grossly underestimated, but it is easy to see that most points fall within a band of reasonable accuracy.

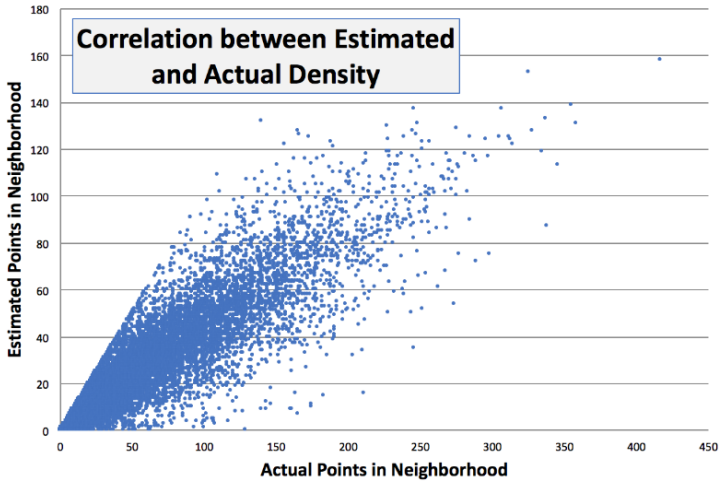


Figure 2. Estimated and Actual Density Correlate Well

Next, view the tables that show how the minimum, mean, and maximum correlation varies by window size and population size. The shaded cells are approximately at the values where correlation is 75%. The bold cells fall on the cells where window size is nearest to √N. on the minimum table, these are the same cells. This supports the conclusion that an acceptable window radius is proportional to √N.

Minimum Correlation per Window Radius

N	20	40	60	80	100	150	200	300	400	500	600	700
1,000	.739	**.799**	.828	.855	.855	.924	.965	.965	.965	1	1	1
2,000	.663	**.834**	.885	.916	.936	.936	.947	.957	.957	.957	.957	.968
4,000	.598	.721	**.774**	.807	.848	.874	.885	.885	.885	.911	.916	.928
6,000	.58	.678	.735	**.784**	.81	.856	.872	.883	.915	.924	.924	.924
8,000	.559	.647	.709	**.763**	.787	.839	.865	.882	.913	.949	.957	.959
10,000	.521	.623	.689	.731	**.761**	.814	.843	.881	.899	.902	.902	.902
20,000	.45	.552	.621	.66	.693	**.749**	.805	.827	.831	.835	.846	.875
40,000	.451	.554	.617	.648	.671	.724	**.764**	.801	.821	.842	.858	.867
80,000	.509	.6	.639	.664	.683	.719	.739	**.758**	.764	.759	.755	.752

Median Correlation per Window Radius

N	20	40	60	80	100	150	200	300	400	500	600	700
1,000	.843	**.946**	.956	.962	.966	.977	.985	.99	.991	1	1	1
2,000	.718	**.862**	.92	.948	.957	.965	.968	.976	.981	.985	.987	.993
4,000	.644	.753	**.817**	.861	.895	.931	.943	.952	.956	.964	.967	.97
6,000	.597	.702	.762	**.808**	.839	.893	.926	.95	.956	.959	.961	.962
8,000	.611	.702	.763	**.8**	.828	.874	.906	.946	.963	.971	.976	.979
10,000	.619	.705	.758	.791	**.814**	.859	.889	.931	.955	.962	.963	.964
20,000	.604	.684	.728	.758	.781	**.822**	.851	.888	.912	.932	.95	.965
40,000	.596	.671	.713	.74	.758	.793	**.818**	.848	.869	.887	.902	.915
80,000	.61	.678	.713	.735	.75	.777	.796	**.821**	.835	.847	.857	.864

Maximum Correlation per Window Radius

N	20	40	60	80	100	150	200	300	400	500	600	700
1,000	.909	**1**	1	1	1	1	1	1	1	1	1	1
2,000	.764	**.899**	.943	.978	.981	1	1	1	1	1	1	1
4,000	.719	.813	**.863**	.903	.943	.969	.981	.985	.988	.988	.988	.991
6,000	.621	.742	.788	**.838**	.877	.924	.956	.993	.998	1	1	1
8,000	.678	.769	.818	**.844**	.865	.9	.936	.992	.998	.999	1	1
10,000	.71	.781	.809	.824	**.841**	.887	.925	.966	.993	.999	.999	1
20,000	.705	.762	.802	.821	.84	**.871**	.884	.911	.933	.956	.974	.991
40,000	.698	.756	.785	.808	.827	.851	**.872**	.879	.898	.914	.936	.954
80,000	.739	.788	.81	.824	.83	.844	.862	**.883**	.889	.91	.927	.934

Figure 3. Data to Support Window Radius

Variable sized and variable density clusters

The case of varying density was simulated by creating many clusters with the same standard deviation, but different numbers of points, hence different densities. Fifty clusters were created from 127,500 points each having one hundred dimensions. The smallest cluster had one hundred points, the next two hundred, on up to five thousand. SLASH grouped them flawlessly, even though the size and density ratio between the largest and smallest clusters was fifty.

Oddly-shaped clusters

Clusters composed by splicing together long, randomly curving chains of Gaussian balls were generated randomly. SLASH had no difficulty clustering them *after* an adjustment was made to the algorithm. Apparently the sample size necessary to accurately estimate how many clusters there are in the data is dependent on whether clusters are spherical or long and twisty. A low sample size will cause the chain to split into many pieces, often so small that they are considered outliers and mishandled. Since research shows that the first estimate of the number of clusters is seldom reduced by more than a factor of three by the Hilbert index search algorithm, if by sampling the index the estimate of the number of clusters drops too much (such as to one or two!), the index so generated is discarded and the sample size increased until a good estimate of the number of clusters is found.

Overlapping clusters

To exercise the density-based reclustering algorithm, two randomly generated Gaussian clusters were positioned far enough apart so that there was little chance of overlap, then brought gradually closer. As the separation was reduced, the points were clustered and the results recorded. Several outcomes were possible. Sometimes the BCubed score was one, meaning the clustering was perfect. Sometimes the homogeneity was one but the completeness was lower. Such a result is excellent. The clusters remained distinct, but one or both were fragmented into smaller pieces that ought to be put back together. Simple refinements to the algorithm could identify and rectify that. On the other hand, if the completeness was one and the homogeneity low, all the points were grouped into a single cluster. An intermediate outcome is a high value for each (but less than one), meaning the right number of clusters was formed and most points are in the appropriate one, with a few out of place.

The tests specify a percentage overlap. Since a Gaussian distribution technically extends to infinity, here is what this means. In a one-dimensional Gaussian distribution, a substantial fraction of points fall outside one standard deviation from the mean. However, as the number of dimensions increases, points begin to cluster tightly in a shell at a radius dependent on the standard deviation and the dimensionality [13]:

$$R \simeq \sigma\sqrt{D}$$

Thus for one hundred dimensions, if one separates the centers of the clusters by twenty standard deviations, the odds of overlap are very small. To put pressure on the algorithm, a much smaller separation of three standard deviations was considered zero percent overlap. Thus a separation of 1.5 σ was considered fifty percent overlap, etc.

Overlap Percent	Perfect Split	Good Split	Good Over-split	Fair Over-split	Bad Split	Unsplit
0.0	29	0	7	0	4	10
45.0	34	0	11	0	2	3
50.0	29	0	14	0	0	7
55.0	33	0	9	0	3	5
60.0	27	0	20	0	0	3
62.5	36	0	9	0	1	4
65.0	30	3	13	0	1	3
67.5	28	5	10	3	0	4
70.0	18	9	13	4	2	4
72.5	20	16	3	3	3	5
75.0	10	18	3	6	6	4

Figure 4. Without Radius Shrinkage Merges

The results show that zero percent overlap does occasionally overlap. The test was repeated fifty times for each percentage on a population of 5,000 points divided into two clusters each having 100 dimensions. The columns Perfect Split, Good Split and Good Over-split constitute successes. In all cases, a majority of trials produced a good result. These first results do not include the effect of radius shrinkage merges. If the merger of two nearby clusters creates a single cluster whose radius is less than the sum of the radii of the two clusters, then those clusters can be merged. Using a shrinkage ratio of 0.6, this is what we see:

Overlap Percent	Perfect Split	Good Split	Good Over-split	Fair Over-split	Bad Split	Unsplit
0.0	38	0	0	0	0	12
45.0	49	0	0	0	0	1
50.0	47	0	0	0	0	3
55.0	40	0	0	0	0	10
60.0	47	0	0	0	0	3
62.5	44	0	0	0	0	6
65.0	43	0	0	0	0	7
67.5	36	2	1	0	0	11
70.0	6	3	0	0	0	41
72.5	1	0	0	0	0	49
75.0	0	1	0	0	0	49

Figure 5. With Radius Shrinkage Merges

The effect of radius shrinkage merges is to cause most runs to move toward either Perfect Split or Unsplit. When the overlap is low, most results are perfect. However, once the overlap gets to 70%, it flips the other way. It may mean that the shrinkage threshold should be lowered. Nevertheless, for most situations, it improves the accuracy significantly.

Scalability Experiments

Timings were taken for end-to-end runs of SLASH, as well as for two key components: the distance measurement and the Hilbert transform. All tests were run on a four-processor machine. For systems with more processors and/or registers, additional loop-unrolling would benefit the distance computation.

When testing SLASH, three of the four key measures of problem size were held constant while the fourth varied. Those measures are population size N, dimensions D, bits per coordinate B, and number of clusters K. The test data consisted of randomly chosen Gaussian clusters where the axes were permitted to vary in radius independently.

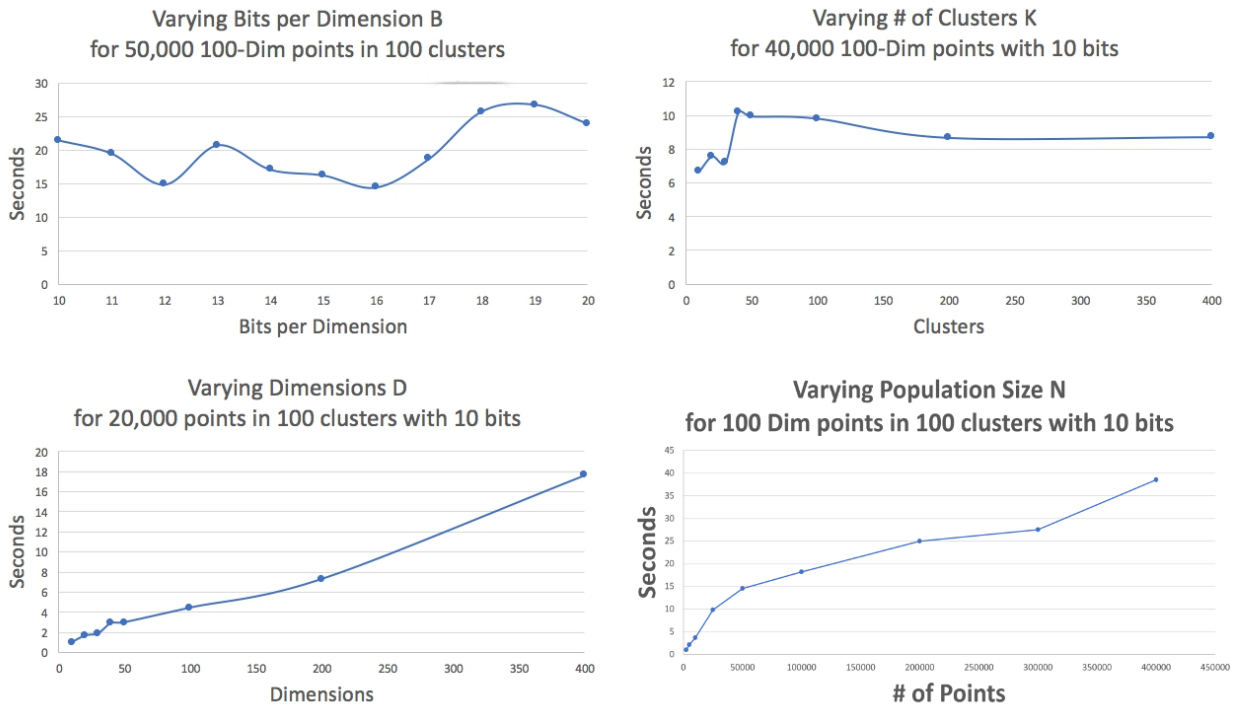


Figure 6. Scalability of Clustering

How are we to interpret these results? Execution times were averaged over many runs, but the random data is regenerated each time. The distribution of the data and the concordance of the Hilbert index can cause the first and most efficient phase to succeed in doing most of the work or leave some for the later phases to do. Looking at the broad trends, SLASH is more insensitive to bits B than expected. The dip in time at 12 and 16 bits could be because of the

machine word size: twice twelve is twenty-four, which is three bytes, while sixteen bits is two bytes.

The dependence on cluster count K looks counter-intuitive at first. When K is small, the odds of finding a Hilbert Index with high concordance (i.e. low fragmentation) becomes high, permitting the efficient first phase to do almost all of the work. Beyond a certain point, increasing K seems to have no effect.

The curve for N requires explanation. For lower numbers of points and a fixed number of clusters, the later phases of the algorithm dominate, and they have poorer performance characteristics. When the number of points increases, more of the work can be done in the efficient first phase, that makes the heaviest use of the Hilbert curve. All in all, the program scales very well with N. In the largest test conducted so far, 100 thousand 1000-dimensional points in one hundred clusters were classified correctly in three minutes on a Windows Surface tablet.

Finally, as promised, there is a clear linear dependence on dimensions D, the most important result from all the testing.

Distance Measurement

The Euclidean distance measure benefited from several optimizations. Here is a comparison between several versions, showing what each refinement contributed. The time is given for one hundred thousand distance comparisons between points with two thousand (2000) dimensions. The times do not include the triangulation optimization for comparing distances. The dot product method is where the magnitudes of the two vectors (their distance from the origin) are computed outside the loop and reused.

Simple Loop	2.409 seconds	–
Loop Unrolling	0.585 seconds	75.71% improvement
Dot Product	0.275 seconds	88.58% improvement

When using distances from points to reference points to triangulate, the performance improvement depends upon where you put the reference points, and the addition of extra points leads to diminishing returns. Currently, the use of fourteen triangulation points seems optimal, and leads to an average case where it can determine whether two points are nearer or farther than the linkage distance L 50% of the time, and the neighborhood distance 70% of the time.

Hilbert Transform

The accompanying charts show how the execution time for the Hilbert transformation varies linearly with bits B, dimensions D, and population size N.

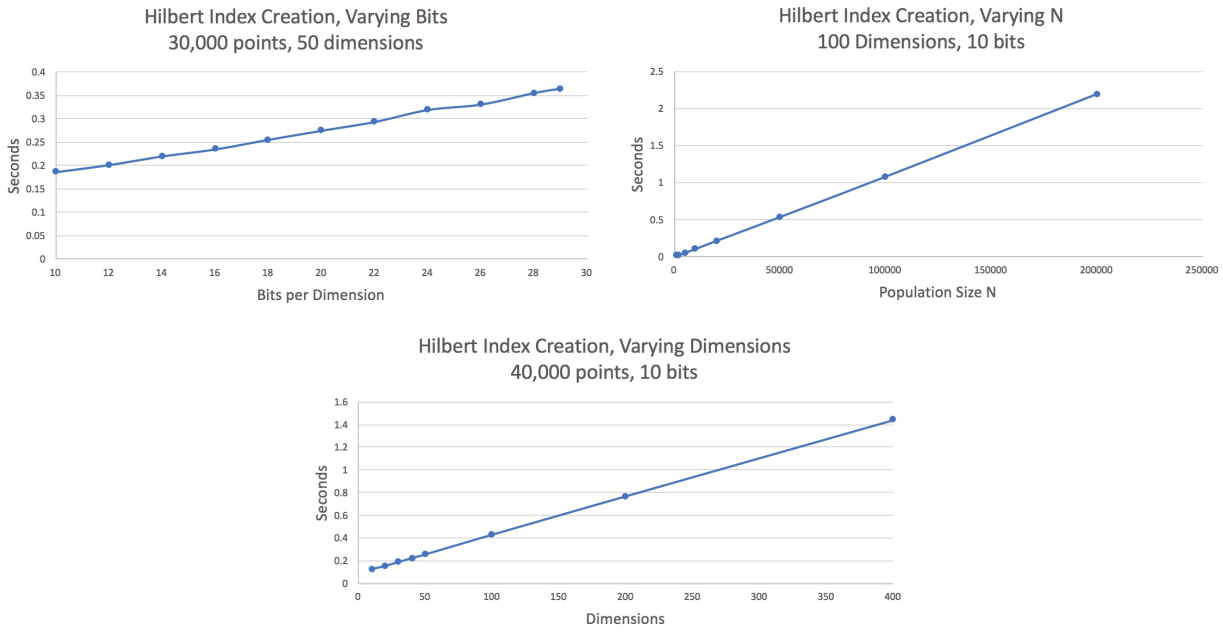


Figure 7. Scalability of Index Creation

Memory Usage

The Hilbert transformation and sorting necessary to form multiple indices consumes the most memory. The permuted version of the data equals the data size and one copy is made for each parallel index tested. Furthermore, the Hilbert index, a BigInteger, is similar in size to the data being sorted and clustered. If four processors are used to search for a good index, the additional overhead can equal eight times the data size.

This problem is solvable, and experiments show that this memory overhead can be reduced to one-tenth of the data size (an eighty-fold reduction!) without sacrificing speed, and in some cases improving the speed. Accomplishing this required five discoveries:

1. Which data unnecessary for clustering could be discarded.
2. A sort algorithm that consumes and disposes of temporary data at a lower, controllable rate while remaining in-place, cache-friendly and parallelizable.
3. A low-resolution Hilbert transform that sort points in a manner consistent with high-resolution transforms while requiring significantly less memory.
4. A point transform that maximizes the utility of low-resolution Hilbert transforms.
5. A means of estimating the precision of Hilbert curve necessary at each phase of the sorting without actually performing the transform.

Earlier attempts at a clustering algorithm needed the Hilbert position at several stages, hence it had to be preserved; the current algorithm only depends on the data points being sorted in Hilbert order. Once the sorting is done, the position (a BigInteger) can be discarded, as can the permuted form of the point. That means that after sorting is done we can reduce memory, but what about during sorting?

The simplest approach that reduces memory is to perform a quicksort that applies the Hilbert transform to each point as it is being compared during the sort operation, then throws the BigInteger away until the next comparison is made. However, in our sort, the comparisons are the biggest expense, not the swaps, so this would slow the sort down terribly.

The next approach considered was to use a distribution sort. Choose many random pivots (instead of the single pivot used by quicksort) and compute the Hilbert position for each, sort the pivots, then distribute the remaining points into the appropriate bucket using a binary search. Since some buckets will receive many points and some few, if we are to throttle the memory usage at each stage, we need to choose the number of buckets carefully. Random trials showed that a number of pivots equaling twice the square root of the number of points yielded the best results. If any buckets received too many points, tests would be done to check for duplicates (using hash code comparisons). In the absence of many duplicates, the large buckets would be further divided by a recursive application of the distribution sort, while the small buckets would be sorted using quicksort. At no level of the recursion would more than $2\sqrt{N}$ Hilbert positions need to be present in memory at once. (The .Net OrderBy only computes the sort key once per item and stores the association until the sort is complete.) However, because of the recursive nature and the disposal of Hilbert positions, on average each point would need to be transformed 2.2 times. This significantly improves upon what happened in the simplest algorithm when you discard each position immediately, but unlike quicksort, because we cannot predict how many items will end up in each bucket, this cannot be done in-place. We can do better.

At this point, one insight was crucial: if you sort points by a low-resolution Hilbert transform that only uses a single bit for each dimension, the ordering will be consistent with that which proceeds from using a high-resolution transform derived from all the bits. If two points have different low-res transforms, their relative order will be preserved. If those points have matching low-res transforms, they will need to be re-sorted with a higher precision transform.

Consider data where the largest values are about one million. Twenty bits of precision are necessary to transform the point. If you perform an initial sort with one bit per dimension, one-twentieth the memory is required. After the first round of sorting (in-place!) one scans the points and groups them according to whether their low-res Hilbert positions match or don't match, before performing a recursive quicksort (in-place!) against each smaller segment, until each segment of the array has only one point with the given Hilbert position or the full resolution of transform is performed and the points are discovered to be duplicates.)

In this algorithm, each point may have a Hilbert transform performed multiple times, but the first round it is only one bit per dimension, the next round maybe seven or eight bits, until finally the full number of bits are required. At each level of recursion more points drop out and do not need to be retransformed. The amortized number of bits of transform required varies based on the characteristics of the data. The experiments that were performed showed this ratio of the number of bits transformed to the full precision to vary from 0.5 to 1.5 for clustered data. (Compare this to 2.2 for the distribution sort!) For totally random data, as little as one-twentieth the memory is needed, since the sorting is done after the one-bit round! Furthermore, the time to perform the transform is linearly proportional to the number of bits, so the sort actually is faster.

The ability of a low-res transform to divide points into small buckets is contingent on whether the median value for each coordinate is near the middle value halfway between zero and the maximum value for the number of bits required. This discovery suggested that the points should be preprocessed by computing the median of each dimension and shifting the values for each dimension independently. This permits the one-bit transform, which slices each dimension in half to also slice the number of points in half. If each dimension divides the points in half, it doesn't take many dimensions before each point is in a bucket by itself – if the dimensions are truly independent, which is true for random data but not clustered data. With clustered data, one ends up with each cluster being in one or a few buckets.

The last link in the chain is deciding at each level of recursion how many bits of precision to use for the next level. This varies for each segment of data. Choose too low and a level of recursion may not divide the points at all, wasting time. Choose too high and unnecessary memory and processing time will be wasted. The key to this is measuring the coarseness of the grid that each number of bits represents. By selecting random pairs of points and testing if they will end up in the same grid cell or not we can estimate the coarseness. If all selected pairs will end up in the same cell, the coarseness is one. If none match, the coarseness is zero (ideal). If the coarseness is 1/K we can expect that number of bits will divide the points into roughly K even buckets. The test consists of iterating over all dimensions and seeing if any multiple of the power of two corresponding to the grid cell width falls between the values taken from the two points. If such a divisive grid line can be found for even one dimension, the points will be separated, otherwise they will not. Experiments show that if a bucket has N points, 2N random pairs will generate an adequate estimate of the coarseness.

Given this sorting algorithm, what characteristics of the data affect its performance? The most important quality is the compactness of the clusters. If the standard deviation of points from the center of their cluster is low, the cluster is compact and more bits are required to sort them. The second most important quality is the number of clusters in the data. If there are more clusters, a low-res transform will split points into more pieces, and later phases will proceed more efficiently, having smaller chunks to sort. If there are few clusters, the initial pass will not divide points much and subsequent passes will require more memory and time.

As figure 8 shows, the variation from a standard deviation per coordinate of twenty to two thousand cause the relative cost to drop from 1.5 to about 0.6, while increasing the number of clusters from ten to one hundred dropped it the rest of the way to 0.5. (The effects of this sort are not represented in the timings elsewhere in this paper as they are a late discovery. The impact on the dependence on N should be substantial, as artificially limiting the number of processors will no longer be required for the large cases. Furthermore, even larger cases will now be possible.)

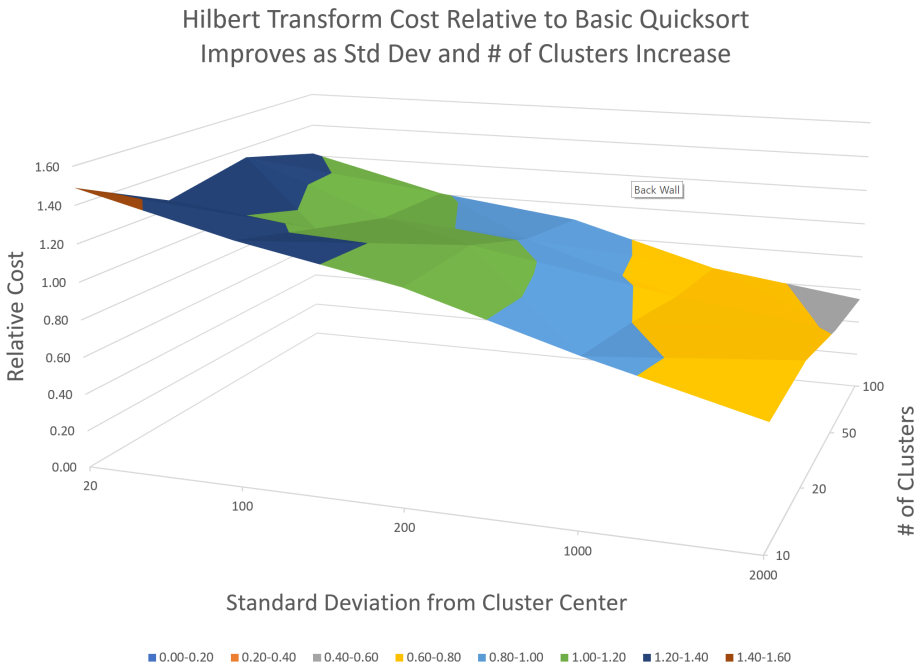


Figure 8. Novel Sorting Algorithm reduces Cost of Hilbert Transform

Future Research

The future of SLASH is to become bigger, brisker, better and broader. The *bigger* part is refactoring it to use a database. It is currently limited to problems with 100,000 points and 1000 dimensions due to memory limitations, or more dimensions and fewer points, or more points and fewer dimensions.

The *brisker* part is to speed up the weak links in the process. The first priority is to triage which third stage clusters are likely to need to be broken apart by the fourth stage, so that the density-based reclustering is only called when needed. This entails characterizing its shape as nearly spherical (not needing division) or oddly shaped. A second priority is performing more efficient density estimation, which would speed up the reclustering. Thirdly, if SLASH chose a more carefully its set of triangulation points, the Euclidean distance comparison might be made almost entirely insensitive to D, which would have ripple effects throughout the whole program.

One of the steps requiring sorting could be replaced with a faster single pass estimator instead of a costly sort. Discovering the linkage distance L could be done differently and based upon the pair distances at various percentiles. A recent research paper [21] explains how one can derive good estimates of quantiles in a single pass by storing only two values: the estimate following the previous sample and the direction and step size of the last adjustment. As each new value comes in, it is compared to the running estimate, a random number is drawn, and a decision made whether to increase or decrease the estimate. When the value consistently moves in one direction, the step size is increased. When it overshoots, the step size is reduced. This approach would potentially lead to less accurate estimates for K, so study is needed.

Making SLASH *better* means handling more edge cases. The use of complete link or average link in place of single link might offer improvements. A new approach to PCCP that can handle irregular shapes would improve the secondary clustering by finding opportunities to merge that are currently overlooked. Also, noise filtering beyond merely randomly sampling the points is an important challenge to tackle. Shared Nearest Neighbor (SNN) as proposed by Ertöz et al [10] to improve the density calculation and prune connections would permit SLASH to keep abutting clusters of different densities apart.

Another way to improve SLASH is to reduce the memory usage of each index. Once constructed, the Hilbert-sorted positions (integers) of the original points could be stored and the Hilbert points discarded. Such an index would occupy a memory footprint proportional to N, but independent of D. This would permit larger cases to be treated in memory. With smaller indices, we could employ more of them. The optimizer currently seeks the index that yields the least fragmentation of clusters. Alongside that we could also seek an index that has the fewest adjacent point pairs in common with the first index. Such an index should be more likely to include links between clusters that were fragmented by the first index. This would permit the efficient Hilbert index-guided single-link phase to do more of the work that would have fallen to the less-efficient later phases.

More research might identify better default values for the parameters that control SLASH, making it easier to use, faster and more accurate. One critical parameter is the unmergeable size needed by the density-based phase. If noise and overlapping clusters cause all or most of the points to be joined into a single massive cluster in the early phases, then the unmergeable size should be set lower so it can split the blob into more pieces.

Finally, SLASH could *broaden* its impact by offering the option to generate hierarchical classification. Also, the primary phase could be used to estimate K and find an initial set of cluster centers for feeding into K-MEANS. If SLASH determines an average cluster size, it can select more starting centroids from large clusters and fewer centroids from small clusters. With a better initial set of centroids, K-MEANS will converge quicker. As far as density-based clustering algorithms are concerned, the density estimates that come from using the Hilbert curve may improve them as well.

Conclusion: SLASH is Accurate, Scalable and Might Improve Existing Algorithms

SLASH is a clustering algorithm that can scale linearly with the number of dimensions, handle oddly-shaped clusters of different sizes and densities, and which requires little tuning to configure. These benefits make it worthwhile to use with high-dimensional data. Equally appealing is the fact that, apart from the Hilbert curve transformation logic (which admittedly is magic), none of the code that makes up SLASH is challenging to understand or implement. It can be extended with modest effort, and several of the phases could be replaced by alternate algorithms. Lastly, there is a real possibility that it could complement popular algorithms like K-MEANS already in wide use with well understood characteristics, further encouraging its adoption.

All source code for SLASH is available on Github.
https://github.com/paulchernoch/HilbertTransformation

References

[1] Yun Tai Lu, "An Efficient Hilbert Curve-Based Strategy for Large Spatial Databases", National Sun Yat Sen University, 2003. The taxonomy is on p4. The authors of this paper make a very different use of the Hilbert curve in their work on database clustering and advertise poorer scalability.

[2] A. M. Fahim, G. Saake, A. M. Salem, F. A. Torkey, and M. A. Ramadan, "K-Means for Spherical Clusters with Large Variance in Sizes", International Journal of Computer, Electrical, Automation, Control and Information Engineering Vol. 2, No. 9, 2008. One must alter K-MEANS significantly to handle clusters of different sizes and densities and to estimate K. Even so, the resulting clusters remain mostly spherical; irregularly shaped clusters are not handled well.

[3] S. Berchtold, D.A. Keim, and H. Kriegel, "The X-Tree: An Index Structure for High-Dimensional Data", Proceedings of the 22nd VLDB Conference, Mumbai, India, 1996.

[4] J. Goodman, J O'Rourke, P. Indyk (Ed.), "Chapter 39: Nearest Neighbors in high-dimensional spaces", Handbook of Discrete and Computational Geometry (2nd Ed), CRC Press, 2004.

[5] J.K. Lawder, "Using State Diagrams for Hilbert Curve Mappings", Technical Report no. JL2/00, August 15, 2000. Page 14 gives this lower bound for no. of states in the state model for n-dimensions:

$$n2^{n-1}$$

[6] C.C. Aggarwal, A. Hinneburg, and D.A. Keim, "On the Surprising Behavior of Distance Metrics in High Dimensional Space", ICDT 2001, LNCS 1973, pp 420-434, Springer-Verlag.

[7] B. Moon, H.V. Jagadish, C. Faloutsos, and J. Saltz, "Analysis of the Clustering Properties of the Hilbert Space-filling Curve", IEEE Transactions on Knowledge and Data Engineering, vol 13, No. 1, January/February 2001.

[8] D.W. Scott and S.R. Sain, "Multi-dimensional Density Estimation", Elsevier Science (preprint), 2004.

[9] N. Kouirokidis and G. Evangelists, "The Effects of Dimensionality Curse in High Dimensional kNN Search", Proceedings of the 15th Panhellenic Conference on Informatics, IEEE, 2011.

[10] L. Ertöz, M. Steinbach, and V. Kumar, "Finding Clusters of Different Sizes, Shapes, and Densities in Noisy, High Dimensional Data," Proceedings of the Second International SIAM Conference on Data Mining, 2003. Page 3 describes why the triangle inequality does not hold true in high dimensions. This paper has a rich discussion of alternate ways to handle density variations. However, the time and memory requirements illustrate problems with the scalability of graph reachability approaches.

[11] M. Ward, "Performance Comparison of the K-means Algorithm", Center for Data Science, New York University. Shows worse than linear increase in time for K-means as k increases.

[12] Y. Tao, K. Yi, C. Sheng and P. Kanos, "Efficient and Accurate Nearest Neighbor and Closest Pair Search in High-dimensional Space", ACM Transactions on Database Systems (TODS), Vol. 35, Issue 3, 2010. Discusses difficulty of choosing the neighborhood radius posed by varying densities, considerations for adapting algorithms for use in databases as well as the closest pair problem.

[13] J. Hopcroft and R. Kannan, "Computer Science Theory for the Information Age", Chapter 1, 2012.

[14] C.D. Manning, P. Raghavan, and H. Schütze, An Introduction to Information Retrieval, Online Edition, Chapter 17, "Hierarchical clustering", Cambridge University Press, 2009. Asserts that because complete-link and average-link (aka group-average agglomerative clustering) are not "best-merge persistent", their execution time is $O(N^2 \bullet \text{Log } N)$.

[15] C. Ding and X. He, "K-means Clustering via Principal Component Analysis", Proceedings of the 21st International Conference on MAchine Learning, Banff, Canada, 2004.

[16] J. Friedmand and J. Tukey, "A Projection Pursuit Algorithm for Exploratory Data Analysis", SLAC-PUB-1312, September 1973.

[17] J. Skilling, "Programming the Hilbert curve", Proceedings of the 23rd International Workshop on Bayesian Inference and Maximum Entropy Methods in Science and Engineering, American Institute of Physics, 2004.

[18] Doddi, Marathe, Ravi, Taylor, Widmayer, "Approximation Algorithms for Clustering to Minimize the Sum of Diameters", Nordic Journal of Computing, 2000.

[19] S. Savaresi , D. Boley, S. Bittanti, and G.Gazzaniga, "Choosing the cluster to split in bisecting divisive clustering algorithms", Technical Report, Department of Computer Science and Engineering, University of Minnesota, 2000.

[20] A. Rosenberg and J. Hirschberg, "V-Measure: A conditional entropy-based external cluster evaluation measure", Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning, pp 410-420, Prague, June 2007, Association for Computational Linguistics, 2007.

[21] Q. Ma, S. Muthukrishnan, M. Sandler, "Frugal Streaming for Estimating Quantiles: One (or two) memory suffices", arXiv:1407.1121, 2014.