Near Neighbor Search in Large Metric Spaces

Sergey Brin*
Department of Computer Science
Stanford University
sergey@cs.stanford.edu

Abstract

Given user data, one often wants to find approximate matches in a large database. A good example of such a task is finding images similar to a given image in a large collection of images. We focus on the important and technically difficult case where each data element is high dimensional, or more generally, is represented by a point in a large metric space and distance calculations are computationally expensive.

In this paper we introduce a data structure to solve this problem called a GNAT – Geometric Near-neighbor Access Tree. It is based on the philosophy that the data structure should act as a hierarchical geometrical model of the data as opposed to a simple decomposition of the data that does not use its intrinsic geometry. In experiments, we find that GNAT's outperform previous data structures in a number of applications.

Keywords – near neighbor, metric space, approximate queries, data mining, Dirichlet domains, Voronoi regions.

1 Introduction

A metric space is a space with an associated distance function which obeys certain simple properties such as the triangle inequality (see Section 3). Hence metric spaces are a very general concept and can be applied to vectors (for example, under Euclidean distance) as well as objects like strings and graphs which cannot be easily represented as vectors (if at all). Finding near neighbors in a metric space refers to selecting the elements of a data set (a finite subset of the space) which are within a certain distance of a given point.

The problem of finding the near neighbors in a large data set has been studied well and has a number of good solutions, *if* the data is in a simple (e.g. Euclidean), low-dimensional vector space. However, if the data lies in a large metric space the problem becomes much more difficult. By a *large* metric space we mean a metric space such that the volume of a ball grows very rapidly as its radius increases. High dimensional vector spaces are an example (a ball of radius 2 in a 20 dimensional Euclidean space is over a million times larger in volume than a ball of radius 1).

Consider the following examples as a small sample of where the problem of finding near neighbors in a large metric space occurs:

Genetics – Finding similar DNA or protein sequences in one of a number of large genetics databases.

Speaker Recognition – Finding similar vocal patterns (e.g., under Fourier transforms) from a database of vocal patterns.

Image Recognition – Finding images similar (using the Hausdorff metric [HKR93]) to a given one from a large image library.

Video Compression – Finding the image blocks of a previous frame that are similar to blocks in a new frame (using a simple L_1 or L_2 metric, possibly after a DCT transform) to generate motion vectors in MPEG video compression.

Data Mining – Finding approximate time series matches (e.g., stock histories or year long temperature).

*Supported by a fellowship from the NSF and by an equipment grant from Digital Equipment Corporation.

Proceedings of the 21st VLDB Conference
Zürich, Switzerland, 1995

Information Retrieval – Finding documents related to a given document in a digital library.

Copy Detection – Finding sentences similar to a user's query in a large database of documents.

All of the examples above fit into the model of finding near neighbors in a large metric space. In particular, the speaker recognition, video compression, information retrieval, and possibly data mining examples find near-neighbors in a high dimensional vector space under the L_1 or L_2 metric[1] (more sophisticated metrics could be imagined).

The remaining examples do not fit nicely into any vector space but since we are working with the more general concept of a metric space we will be able to address them. The genetics and copy detection examples above find near neighbors in the metric space of strings under some *edit* distance function. The image recognition example does not fit into either a vector space or a string space but it still qualifies as near-neighbor search in a large metric space.

Every data type above has some degree of correlation in its distribution. While that correlation may be small (i.e., the data resembles random vectors), it must be exploited to get good performance in a near neighbor search. To do this in an application independent manner requires that the data structure capture the intrinsic geometry of the data (Section 3). As we will see (Section 4), our data structure, the GNAT, captures the geometry of data collections such as the ones mentioned above by hierarchically breaking it down into regions which try to preserve fundamental geometric structure.

2 Related Work

A large amount of work has been done to solve specific instances of near-neighbor finding problems. Numerous articles have been written regarding finding similar vectors (e.g., time-series and geographic data), text (files and documents), images, sounds (word recognition), etc. A more limited but still substantial amount of work has addressed the general problem of finding near neighbors in an arbitrary metric space[2]. This work has mostly fallen into two categories. In one category, we assume that distance calculations are so expensive that even an $O(n)$ or $O(n \log n)$ search algorithm is acceptable as long as it reduces the number of distance calculations. This is the case as long as the database size is fairly small compared to the range of the search [FS82] or if preprocessing is not allowed and only arbitrary pre-computed distances are given [SW90]. For the examples mentioned in Section 1 neither of these hold and, while distance computations are expensive, the $O(n)$ cost of such an algorithm would dominate for a large data set.

The other category of solutions are hierarchical and typically have an $O(\log n)$ query time *given a sufficiently small range* (typically too small to be practical). They are of the following form: The space is broken up hierarchically. At the top node, one or several data points are chosen. Then the distance between each of these to each of the remaining points is computed. Based on these distances, the points are separated into two or several different branches. For each branch, the structure is constructed recursively.

J. K. Uhlmann outlined the foundation for two different methods, more generally referred to as metric trees [Uhl91]. One of these methods, subsequently called vp-trees,[3] was implemented by P. N. Yianilos [Yia93]. The basic construction of a vp-tree is to break the space up using spherical cuts. To build it, pick a point in the data set (this is called the *vantage point*, hence the name *vp*-tree). Now, consider the *median* sphere centered at the vantage point with a radius such that half the remaining points fall inside it and half fall outside. For every other point, put it in one branch if it is inside the sphere and in another branch if it is outside the sphere. Now, recursively construct the lower level branches.

This approach has benefits in that it requires only one distance calculation when visiting a node during a search and it automatically creates balanced trees. However, it suffers from regions inside the median sphere and outside the median sphere being extremely asymmetric, especially in a high-dimensional space. Since volume grows rapidly as the radius of a sphere increases, the outside of the sphere tends to be very thin, given that there are as many points on the inside as on the outside. This limits the amount of pruning that can be done during a search and reduces performance. In our work, we try to avoid such asymmetries. While the limited branching factor of 2 can also be viewed as a weakness, we have conducted experiments with higher degree variations of vp-trees and find little improvement in performance (see Section 5.2).

The other method, a *generalized hyperplane tree* (gh-tree), is constructed as follows. At the top node, pick two points. Then, divide the remaining points based on which of these two they are closer to. Now, recursively build both branches. This method is an improvement in that it is symmetric and the tree struc-

[1] Recall that $L_k(\overline{x}, \overline{y}) = \sqrt[k]{\sum |x_i - y_i|^k}$. So L_1 is the familiar *city block* distance and L_2 is the Euclidean distance.

[2] Some of the papers we mention below address the problem of finding *nearest* neighbors. However, their methods can be applied to finding all near neighbors with minimal change.

[3] We do not look at the enhancement of vp-trees called vp^{sb}-trees.

ture still tends to be well balanced (assuming sufficiently random selection of the two points). However, it has a weakness in that it requires two distance computations at every node during a search and is limited to a branching factor of two.

An improved variation of gh-trees was implemented at ETH Zurich [BFR$^+$93] as *monotonous bisector trees* (sic), MBT's, to deal specifically with text. However, nothing in the method would have prevented them from dealing with arbitrary metric spaces. The key difference between MBT's and gh-trees is that MBT's only select *one* new point at each new node. They do this by reusing the point they are associated with in the parent node. As a result, MBT's overcome the first weakness but the branching factor remains a problem.

The most relevant works, however, are also the oldest. Burkhard and Keller suggested several data structures (and algorithms) [BK73] for approximate search. The first is very similar to vp-trees except that it requires a finite number of discrete distance values such as in string edit distance. Essentially, for every vantage point, a separate branch is allocated for every possible distance value. This method, however, suffers from the same asymmetry problem as the vp-trees.

The other two data structures, which are the closest to the GNAT, break up the space into a number of balls, storing the radii and centers. More specifically, divide the data points into groups using some method (this was left as a parameter). Pick a representative of each group and call it the center of the group. Then, calculate the radius (the maximal distance to another point) from the center for each group and pruning of searches is performed based on these radii. Recursion is briefly mentioned but not analyzed.

The third method, an enhancement of the second, additionally requires that the diameter (the maximal distance between any two points) of the points in any group be less than a constant, k, and the group is then called a clique. In this case, consider the set of all maximal cliques (those cliques not strictly contained in any other cliques). It must be a cover of the data set since every point is in at least one maximal clique. Then, a minimal subcover is chosen as the set of all groups.

These two schemes act as reasonably good models of the data space they store and if extended to a hierarchical structure, they have an arbitrary branching factor. However, they have several weaknesses. First, they do not work well with non-homogeneous data, since we could easily end up with a lot of cliques containing only one point and several cliques containing very many points. Additionally, distance computations are not fully exploited in that distance to the center of one clique is not used to prune other cliques. Finally, while we do not focus on the cost of preprocessing in this paper, this cost was reported to be extremely high in the third method.

K. Fukunaga and P. Narendra worked out a very similar scheme, which requires more than just a metric space, to create a tree structure with an arbitrary branching factor in 1975 [FN75] as follows. Divide the data points into k groups. (How this is done is left as a parameter of the structure but in tests they used a clustering algorithm which requires more than just a metric space.) Then compute the mean[4] of each group (once again a departure from a metric space) and the farthest distance from that mean to a point in the group. Then recursively create the structure for each group. While this method tends to have nice symmetric properties (given a reasonable clustering algorithm) that reflect the space and it has an arbitrary branching factor, it has several weaknesses. First, it relies on more than just a metric space; second, it requires many distance computations at each node and does not use them fully; and third, it does not deal effectively with balancing.

In this paper we present GNAT's which can be viewed as both a generalization of Fukunaga's method and a generalization of gh-trees. GNAT's provide good query performance by exploiting the geometry of the data. Unfortunately, while query time is reduced, the build time of the data structure is sharply increased. However, if the application is query dominant (or even if there are roughly as many queries as data points) the relative cost of building a GNAT becomes negligible. In tests, we find that GNAT's almost always perform better than both vp-trees and gh-trees, and scale better.

3 Large Metric Spaces

To formalize the problem, we present the standard definitions of a metric space and near neighbors.

Definition 1 (Metric Space) *A* metric space *is a set X with a distance function d: $X^2 \rightarrow \mathcal{R}$ such that: $\forall x, y, z \in X$,*

1. *$d(x, y) \geq 0$ and $d(x, y) = 0$ iff $x = y$. (Positivity)*
2. *$d(x, y) = d(y, x)$. (Symmetry)*
3. *$d(x, y) + d(y, z) \geq d(x, z)$. (Triangle Inequality)*

Definition 2 (Near Neighbors) *Given a metric space (X, d), a data set $Y \subseteq X$, a query point $x \in X$, and a range $r \in \mathcal{R}$, the* near neighbors *of x are the set of points $y \in Y$, such that $d(x, y) \leq r$.*

[4] The mean of a set of points (vectors) in a vector space is simply their sum divided by their number. The concept of a mean is not meaningful for arbitrary metric spaces.

Since we are dealing with arbitrary metric spaces, we assume the following model of computation: we are given the data set Y and a "black box" *Dist*, to compute the distance, d, between members of Y. We can pre-process Y (using *Dist*) and then we want to be able to answer user queries quickly. Queries will be of the form (x, r), requesting the members of Y which are within r of x. If r is fixed for all queries then it can be taken into account in the preprocessing phase. For the indexing schemes we present in this paper, however, r can vary from query to query.

The first important observation is that it is impossible to deal efficiently with all metric spaces. In particular, consider the metric space where the distance between two points is 0 if they are the same point and 1 if they are different. Then our only option in finding the near neighbors of a given query point with any range less than 1 is a linear search and no fancy data structure will save us. In fact, the more any space resembles such a metric space, the more difficult it will be to search.

Furthermore, the distribution of the data set in the metric space is more important than the metric space itself. If the data lies on a two-dimensional surface that is embedded in a 50-dimensional space, query times will behave more like that of a two-dimensional space than that of a 50-dimensional space, given an intelligent data structure. In a sense, for a high-dimensional space, the data determines the "geometry" of the space more than the constraints of the space itself do. For example, if we take a random set of words out of a book, we are working in the space of *all* strings (over a certain alphabet), but in this particular case we are much more likely to encounter some strings, like "the", than others, like "xyzzy". Such highly nonuniform distributions of data points will significantly affect search performance.

Since visualizing high-dimensional data is difficult we look at some simple measures to help us understand the geometry of a given data set. A particularly useful measurement is the distribution of distances between random points of the data. While the scales of these distributions vary greatly, we can compare them by considering at what range we would be interested in finding near neighbors. In each of the graphs of the distributions that follow, we have calculated the probability density function (PDF) of a random distance calculation. To determine this we used 5000 random distance calculations from the data set, distributed the distances into a number of buckets, and plotted the relative probability of falling into each bucket. We use relative probabilities so the plot does not depend upon the bucket size and because this is the standard way to visualize a continuous random variable. Since these are not absolute probabilities, they may exceed one (however, the integral of any PDF is 1).

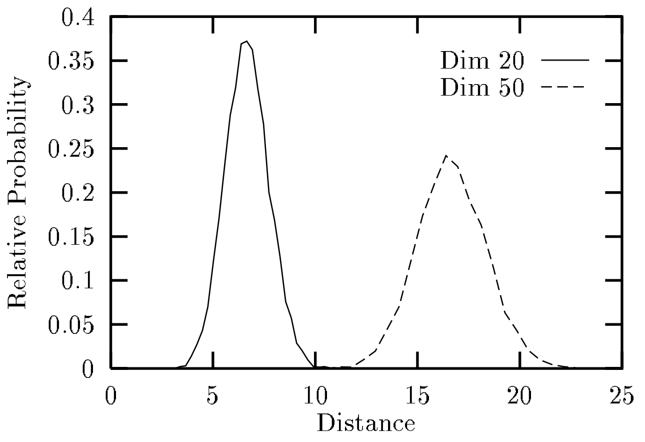

Figure 1: Distribution of distances between vectors chosen uniformly from unit cubes under the L_1 metric in 20 and 50 dimensions.

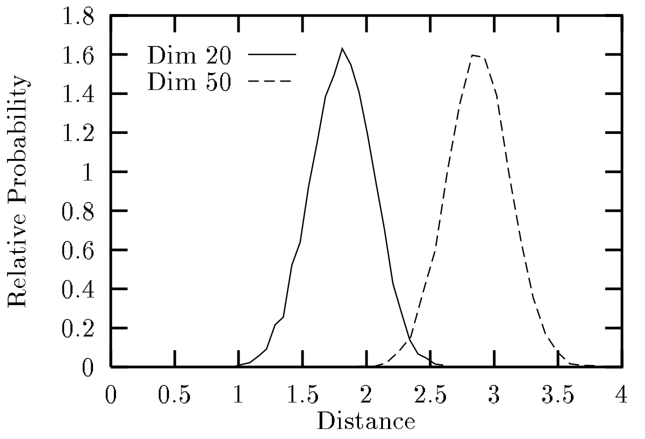

Figure 2: Distribution of distances between vectors chosen uniformly from unit cubes under the L_2 metric in 20 and 50 dimensions.

The distributions of distances between random, uniformly chosen vectors in 20 and 50 dimensional hypercubes of side 1 under the L_1 metric are very closely approximated by Gaussian distributions because of the Central Limit Theorem (Figure 1). For the L_2 metric, we obtain a Gaussian-like (though not exactly Gaussian) distribution (Figure 2). Note that the distributions for 50 dimensions should be viewed in relation to their larger ranges and hence are really quite narrow. The fact that the peaks are narrow indicate that the distance function has low entropy and that it may be difficult to index the data since arbitrary distance measurements will provide us with little information (by contrast a uniform distribution would have high entropy and distance measurements would give us a lot of information).

Correlated data has somewhat different properties and tends to have a much flatter distance distribu-

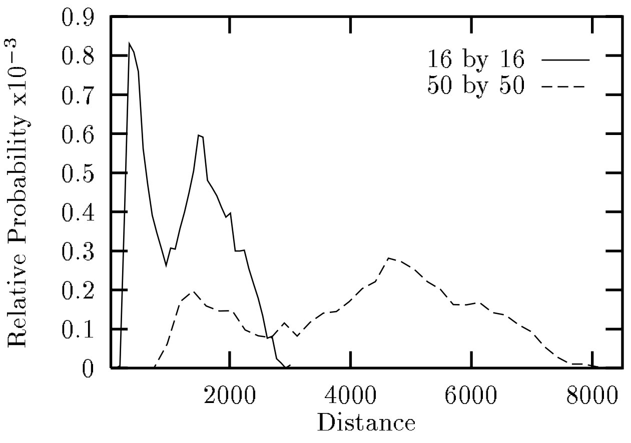

Figure 3: Distribution of distances between 16 by 16 and 50 by 50 subimages under the L_2 metric.

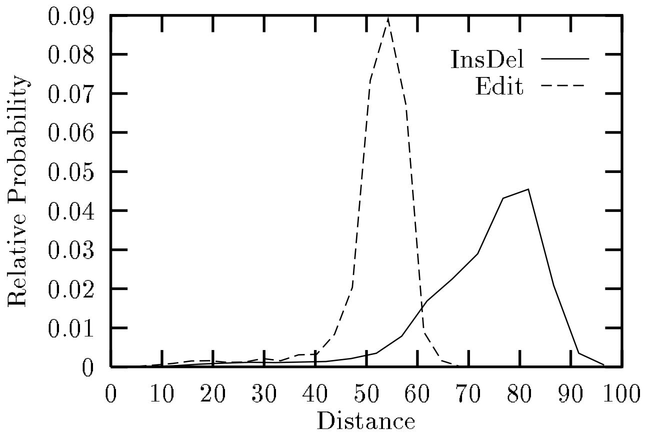

Figure 4: Distribution of edit distances in lines of text.

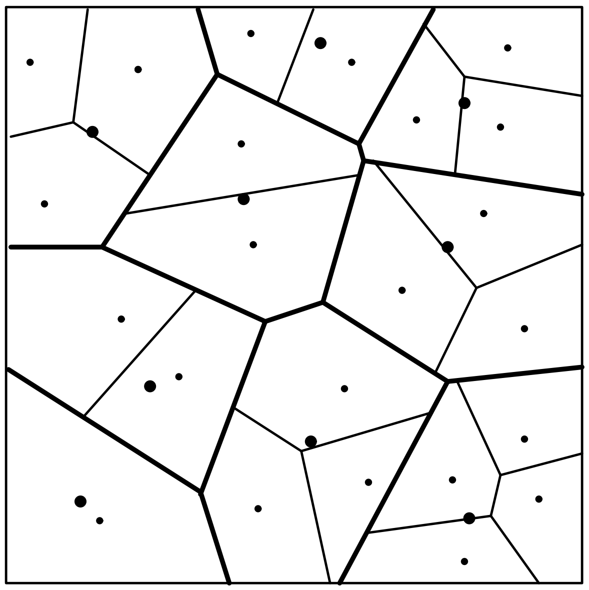

Figure 5: A Simple GNAT (see Section 4)

tion. For example, consider taking random 16 by 16 or 50 by 50 blocks from an image, then treating them as 256 or 2500 dimensional vectors respectively by using a dimension for every pixel. Note that the dimensions corresponding to adjacent pixels should be highly correlated since adjacent pixels have very similar colors very frequently. In essence, the images are using an extremely small portion of their very high dimensional space and we will capitalize on this during our search. Taking the L_2 distances between these image vectors creates a distribution with two major maxima (Figure 3) (the source image was a 640x480 grayscale version of Seurat's "A Day in the Park"). The first maximum, near 0, indicates a great deal of clustering in the data since small distances are so probable; the second major maximum is one that is common to all large metric spaces we will investigate, which indicates that average distances are fairly likely.

As another example, consider taking lines of text (averaging 60 characters) from a large text document (*A Tale of Two Cities* in this example) and using a simple edit distance function.[5] We consider two different such functions. Both count the minimum number of operations needed to get from one line to the other. The first distance function, *InsDel* allows only inserts and deletes of single characters as operations. The second, *Edit*, adds the operation of replacing one character (which previously cost 2 operations - an insert and a delete) and is consistent with what is usually used to compute distance between strings. Both distance functions have the unfortunate property of requiring $O(n^2)$ steps to compute where n is the number of characters, hence minimizing such calculations is very important. When we map out their distributions (Figure 4), both turn out to be considerably less correlated than the images and look roughly like the uniformly chosen vectors (Figures 1 and 2).

4 GNAT's

Our goal when designing GNAT's was to have a data structure that reflected the intrinsic geometry of the underlying data. More specifically, the top node of our hierarchical structure should give us a very brief summary of the data as a metric space, and as we progress down the hierarchy we get a more and more accurate sense of the geometry of the data. This is achieved as a hierarchical, Dirichlet domain[6] based,

[5] Note that there is no valid distance function which will work well for subsequence matching (finding lines of text in the data set which contain a substring similar to a given one) because of the symmetry requirement. To use metric spaces for this, it is necessary to consider the data set to be the set of all the subsequences.

[6] Computer scientists may know these better as the cells of a Voronoi diagram but since we are not referring to the edge and

structure. Given a number of points, $x_1, \ldots, x_k$, the *Dirichlet domain* of x_i consists of all possible points in the space which are closer to x_i than any other point x_j $(j \neq i)$.

At the top node of a GNAT, several distinguished *split* points are chosen and the space is broken up into Dirichlet domains based on those points. The remaining points are classified into groups depending on what Dirichlet domain they fall into. Each group is then structured recursively. A simple example of such a structure is illustrated in Figure 5. The heavier points represent the split points of the top node and finer points are split points of the subnodes. The thick lines represent the boundaries of the regions of space that belong to the top level split points and the thin lines are those of the low level split points.

Another goal of GNAT's is to make full use of the distances we calculate (at least within one node). Therefore, instead of relying on the Dirichlet structure to perform our search, we also prune branches by storing the ranges of distance values from split points to the data points associated with other split points during the build. Then, if query points fall outside these ranges by more than the range of the search, we can prune the split point and everything under it from the search.

For example, suppose we know that all points in the region associated with split point q have distance between 3 and 4 from split point p and we want to search for points within a radius of 0.5 from a query point x. Then if x is less than 2.5 from p, we can apply the triangle inequality (to the points p, x, and y where y is any point in the region of q) and safely prune the node associated with q (see Figure 6; the notation is explained in Section 4.1).

4.1 A Simplified Algorithm

The basic GNAT construction is as follows:

1. Choose k *split* points, $p_1, \ldots, p_k$ from the dataset we wish to index (this number, known as the *degree* can vary throughout the tree, see Section 4.3). They are chosen randomly but we make sure they are fairly far apart (see Section 4.2).

2. Associate each of the remaining points in the dataset with the closest split point. Let the set of points associated with a split point p_i be denoted D_{p_i}.

3. For each pair of split points, (p_i, p_j), calculate the $range(p_i, D_{p_j}) = [min\_d(p_i, D_{p_j}), max\_d(p_i, D_{p_j})]$, a minimum and a maximum of $Dist(p_i, x)$ where $x \in D_{p_j} \cup \{p_j\}$.

4. Recursively build the tree for each D_{p_i}, possibly using a different degree (see Section 4.2).

vertex structure of a Voronoi diagram and are not restricting ourselves to Euclidean spaces, we refer to these cells as Dirichlet domains.

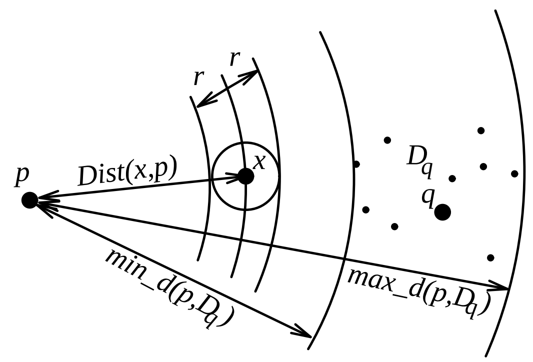

Figure 6: Pruning branches using ranges. Here, we can safely prune D_q since $Dist(x,p) + r < min\_d(p, D_q)$.

A search in a GNAT is performed recursively as follows:

1. Assume we want to find all points with distance $\leq r$ to a given point x. Let P represent the set of split points of the current node (initially the top node of the GNAT) which possibly contain a near neighbor of x. Initially P contains all the split points of the current node.

2. Pick a point p in P (never the same one twice). Compute the distance $Dist(x,p)$. If $Dist(x,p) \leq r$ add p to the result.

3. For all points $q \in P$, if $[Dist(x,p) - r, Dist(x,p) + r] \cap range(p, D_q)$ is empty, then remove q from P.

   We can do this for the following reason. Let y be any point in D_q. If $Dist(y,p) < Dist(x,p) - r$, then, by the triangle inequality, we have $Dist(x,y) + Dist(y,p) \geq Dist(x,p)$ and hence $Dist(x,y) > r$. Alternatively, if $Dist(y,p) > Dist(x,p) + r$, we can use the triangle inequality, $Dist(y,x) + Dist(x,p) \geq Dist(y,p)$ to deduce $Dist(x,y) > r$. The point y cannot fall into the range $[Dist(x,p) - r, Dist(x,p) + r]$ because then $range(p, D_q)$ would intersect it. (See Figure 6.)

4. Repeat steps 2 and 3 until all remaining points in P have been tried.

5. For all remaining $p_i \in P$, recursively search D_{p_i}.

4.2 Selecting Split Points

One of the issues we had to deal with was the selection of split points (step 1 in the construction). We wanted split points to be more or less random, since this way they are likely to be near the centers of clusters. However, we did not want the split points to bunch up, since if they did, distance calculations from them would not have as much information content (distances measured from one member of a bunch would be similar to those of another) and they would not model the geometry of the data well. Furthermore, if several split points were in the same cluster, they would likely divide the cluster at too high a level.

The strategy we settled on is to sample about 3 times the number of split points we want; call these the *candidate* split points. Then pick those candidate points that are the farthest apart (according to a greedy algorithm). The number 3 was arrived at empirically. More specifically this is done as follows:

Pick one of the candidate points at random. Then pick the candidate point which is farthest away from this one. Then pick the candidate point which is the farthest from these two (by farthest we mean that the minimum distance from the two is the greatest). Then pick the one farthest from these three. And so on, until there are as many split points as desired. A simple dynamic programming algorithm can do this in $O(nm)$ time where n is the number of candidate points and m is the final number of split points.

4.3 Choosing the Degree of a Node and Balancing the Tree

For our initial experiments we chose a degree and kept it constant over all the nodes in a tree. This worked reasonably well with uncorrelated data. However, for correlated data, it sometimes unbalanced the tree substantially. Attempts to balance the size of branches (by weighting distances to split points) tended to harm performance more than they helped. So we considered "balancing" the tree by assigning higher degrees to those nodes that contained too many data points. This is done as follows:

The top node is allocated a degree k. Then each of its children is allocated a degree proportional to the number of data points it contains (with a certain maximum and minimum) so that the average degree of a child is the global degree of k. This process works recursively so that the children of each node have average degree k (ignoring the maximum and minimum degrees). The minimum used in tests was 2 unless there only was one point and the maximum was $5k$ or 200, whichever was smaller.

4.4 Space and Time Complexity

Unfortunately, GNAT's are sufficiently complex that they are unwieldy to formal analysis. Therefore the results we present here are weak and limited.

We use the following notation:

- N – the number of data points.
- n – the number of nodes. Empty nodes are not counted.
- s – the maximum size (space requirement) of a data point.
- d – the maximum degree of a node.
- k – the average degree (equal to N/n).
- k_2 – the second moment (the average of the squares) of the degree.
- l – the average depth of a point in the GNAT.
- s – the amount of memory needed to store a data point.

We produce the following simple results:

Space (memory) A GNAT takes $O(nk_2 + Ns)$ space. In practice, k_2 does not turn out to be much higher than k^2.

Preprocessing Time This is the main disadvantage of GNAT's as compared to other data structures. In a perfect scenario, when the tree is completely balanced, we get a preprocessing time of $O(Nk \log_k N)$ distance calculations which is a factor of $k/\log k$ more than binary trees requiring only one distance calculation per node. In the real world, this can be substantially more due to the fluctuating degrees and can be bound only by $O(Nld)$ distance calculations. In tests, l tended to be very close to, though slightly more than $log_k N$ and certainly not all nodes were of degree d so the real preprocessing time lies somewhere between those two extremes.

Query Time This is the most difficult performance attribute to evaluate but it is also the most important. While there have been upper bounds in previous works, they have either been restricted to particular domains or have made assumptions which make them of no practical use. As a result of this and the added complexity of GNAT's we rely on the experimental results (see Section 6).

We do not address how to layout GNAT's on disk and analyse the performance impact of disk based storage in this paper. A major reason for this is that the layout would have to be strongly dependent on the type of data. Recall that during a query, the distances between the query point and the split points it encounters along the way (not including the ones that are pruned) are computed. Hence the non-leaf nodes must contain enough information to compute distances to the split points. This information could easily take up many disk blocks (in the case of 50 by 50 images for example), be variable sized (in the case of strings), or be a pointer to another structure (an (x, y) offset to a large image file). Therefore it is unlikely that a general scheme could work well in all these cases.

5 Implementation

The system used for testing GNAT's and other data structures underwent several major revisions, being implemented in Mathematica, then C, and finally C++. In each of these versions, the benefit of having the data structure rely only on the distance function was a tremendous advantage.

In the final version, the code to handle vectors and text (including generation and/or loading and distance functions) is under a hundred lines each. The code for images is just over a hundred lines, mainly to deal with the file format.

5.1 Data Types Supported

The data types (metric spaces) with which the system works are as follows:

Vectors – The simplest of the data types, these are N-dimensional vectors from a hypercube of side 1 and can be chosen in two different ways – chosen uniformly from $\mathcal{R}^n$, and chosen uniformly from $\mathcal{R}^2$ and then mapped into $\mathcal{R}^n$ using a simple continuous function. Both the L_1 and L_2 metrics are supported.

Image Blocks – These are 16 by 16 blocks chosen randomly from a gray-scale image. Two images were used in tests and they produced very similar results despite being very different - a digitized version of "A Day in the Park" by Seurat and a picture of an SR71 Blackbird. Distance between image blocks is computed very simply (since this is how current MPEG motion vector estimation schemes do this) by considering the blocks as 256 dimensional vectors and using L_1 or L_2 distance. Given that the code can handle arbitrary sized blocks a few experiments were run on 50 by 50 pixel blocks, since they have some interesting clustering properties.

Lines of Text – While the vectors and images both fit into vector spaces, lines of text do not. These were lines taken from a text document. Two different documents were attempted - all five acts of "Hamlet", which unfortunately totaled only about 4000 lines, and Dickens' "A Tale of Two Cities". For Dickens, this was a less meaningful test since taking lines out of a novel is not particularly reasonable (sentences would have been better but they were too long for substantial similarity to occur). Both texts were processed before being read by normalizing whitespace and getting rid of very short lines. Additionally, in Hamlet, speaker names were stripped. Results for these two were not as similar as one might expect (see Section 6). Both the *InsDel* and the *Edit* distance functions (see Section 3) were implemented and tested.

5.2 Data Structures Supported

The final implementation supports a number of data structures including GNAT's.

VP-Trees – See Section 2 for a brief description. In the tests presented here, we did not use any sampling technique to chose vantage points since we could not be sure that we would do it identically to [Yia93]. However, some limited tests with sampling indicated that savings were in the 10% range for images and were negligible for text and random vectors. In graphs, these are labeled as vp$_2$-trees since they are the special case of vp$_k$-trees where $k = 2$.

VP$_k$-Trees – A generalization of vp-trees which differs in that at each node, instead of the remaining data points being split into two halves based on their distance from the vantage point, they are split into k sections of equal size (also based on distance from the vantage point). These were found to perform very similarly to vp-trees (frequently a little better) but there was not a sufficiently large difference to warrant further investigation.

GH-Trees – See Section 2 for a brief description. They are essentially GNAT's of constant degree 2 without the sampling for split points and degree variation throughout the tree. Since they perform worse than GNAT's of degree 2, not many experiments were performed.

OPT-Trees – These use a much smaller number of distance computations for queries than any other structure but they lose out by having far more costly other computations (even super-linear in the number of data points). The idea here is to pick a number of vantage points. Measure the distances from each to all the other points and store these in a table. When a query comes along, measure its distance to the first vantage point and based on that weed out all of the impossible data points. Then take the next vantage point and do the same. Continue until no more data points are pruned. Then, check each of the remaining ones individually. Up to the choice of vantage points this gives more or less the optimal performance, in terms of distance calculations. This structure can serve as a lower bound for distance calculations but is not a realistic goal to shoot for if one wants a scalable structure.

GNAT's – This is the main data structure of this paper, described in Section 4.

For each of these, we check how many distance calculations are used to both build the structure and perform queries.

6 Tests and Results

Given the flexibility of the test-bed system, the number of possible interesting tests to run exceeded by far the number which could be run (in a reasonable amount of time) which in turn exceeded by far the number of test results presented in this section. Note that the benefits of GNAT's varied and in this section we try to present the range of results that were obtained.

Also, while opt-tree plots are in some of the graphs, it is important to keep in mind that these structures are only efficient in terms of distance calculations but are very inefficient otherwise. They are provided just to serve as a lower bound.

6.1 Random Vectors

A number of tests were performed on random vectors. The dimension was set at 50, uniformly chosen vectors were produced, and the L_2 metric was used. The range was varied and a number of different data structures were tested. The number of data points was 3000 (Figure 7) in one test and 20000 (Figure 8) in another. The number of test queries used in every case was 100.

The first thing to note is how difficult it actually is to perform these searches. All of the data structures seemed to struggle with ranges above 0.3 (reasonable queries could easily have ranges considerably above 0.5), looking at more than 50% of the data points in many cases. This is caused by the low information content of the distance calculations since they tend to return very similar numbers (Figure 2).

Despite the difficulty that all these methods have, high degree GNAT's come out far ahead. In particular, the GNAT's of degrees 50 and 100 had more than a factor of 3 improvement over vp-trees in many cases.

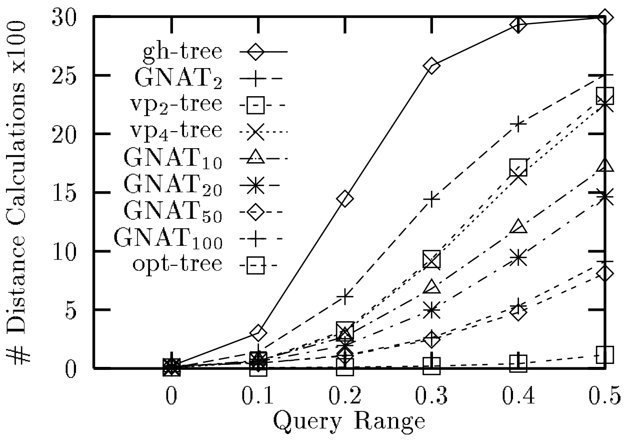

Figure 7: Varying Query Range for 3000 Vectors

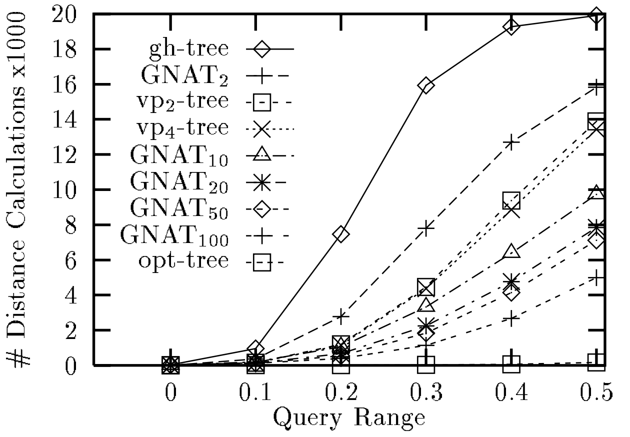

Figure 8: Varying Query Range for 20000 Vectors

6.2 Text

We ran a number of experiments with text with varied success. When testing 3000 lines of Hamlet using the *InsDel* distance, speedups above vp-trees were in the factor of 2 range (Figure 9). Testing on 10000 lines of *A Tale of Two Cities* with the *Edit* distance yielded much more dramatic yet varied speedups, ranging from around 50% with a query range of 10 to more than a factor of 6 with a range of 2 (Figure 10).

6.3 Images

The performance of GNAT's versus vp-trees on images varied greatly but they were not nearly as dramatic as the results for random vectors and text. For example, for 16 by 16 block images and the L_2 metric, the higher degree GNAT's gave only about 15% to 25% improvement over vp-trees (Figure 11). When using 50 by 50 blocks, results were a little better, with improvements in the 15% to 35% range for ranges above 200 (Figure 12). This is a clear indication that more work needs to be done to deal with clustered data.

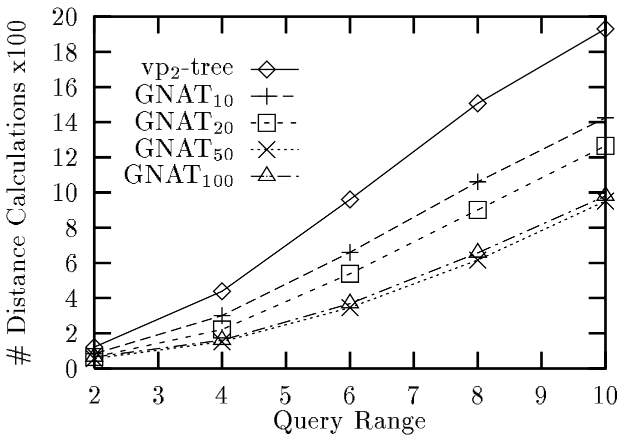

Figure 9: Varying Query Range for 3000 Lines of Hamlet Using the *InsDel* Distance

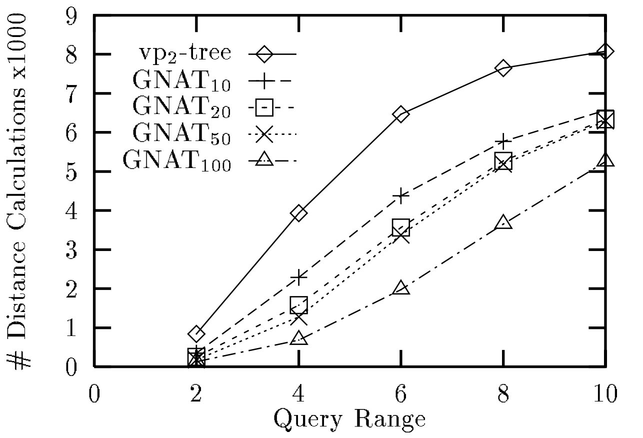

Figure 10: Varying Query Range for 10000 Lines of Dickens Using the *Edit* Distance

7 Conclusion and Future Work

In working with large metric spaces, we have proved our intuition from low-dimensional spaces wrong in many ways. In many cases explored in this paper the data lies in a very large metric space whose only easily recognizable and readily usable structure is the distance between its points. In other words, the space is so large that it is meaningless to consider and use its geometry and one should concentrate on the intrinsic geometry of the actual set of data points.

Consequently, it is important to exploit the constraints of the distribution of data rather than rely on those of the whole space. Therefore, GNAT's try to model the data they are indexing. There are several important issues involved in doing this.

First, does one break up the space by occupation or by population? In other words, if the data is composed of a large cluster and a few outliers, should the top node assign one branch for the cluster and a few branches for the outliers or should it split the cluster immediately and not worry about the outliers until later? In GNAT's we decided to compromise by first sampling (the by-population approach) and then by picking out the points that were far apart (the by-occupation approach) when choosing split points. The current method of selecting points that are far apart can become asymmetric and some pathological behavior was observed (though it didn't impact query performance much). This remains a problem for future work.

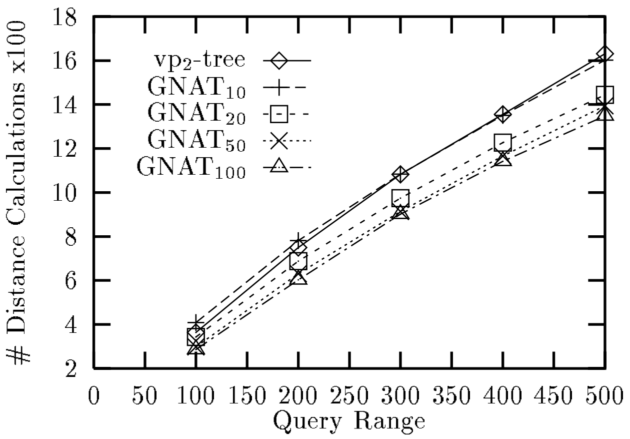

Figure 11: Performance for 3000 16 by 16 images

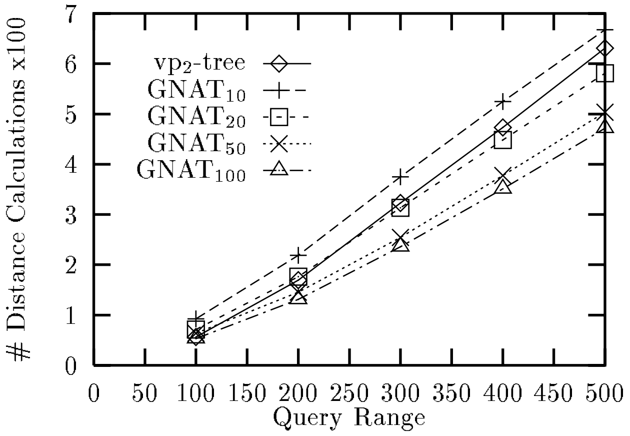

Figure 12: Performance for 3000 50 by 50 images

A second issue is how to handle balancing. In our experiments, we found that good balance was not crucial to the performance of the structure. We attempted to improve the structure by using "weighted" Dirichlet domains but these tended to decrease performance rather than improving it. (They did reduce build time though.) Intuitively, when the tree structure is altered so that it is balanced rather than so it reflects the geometry of the space, searches tend to descend down all branches. As a result we decided to keep the tree depth from varying too much by adjusting the degrees of the nodes.

For future work, we are considering new methods of building the tree. Bottom-up constructions could lead to very good query performance but their $O(n^2)$ construction cost will not scale well. Consequently, we are considering schemes where a top-down construction is used but then is iteratively improved until it converges to a bottom-up type construction.

Another important research direction is to begin to use approximate distance metrics. For example, in order to compute near neighbors in text using the edit distance (an expensive computation), we can first use the q-gram distance [Ukk92] (a relatively fast computation) to narrow the search quickly and then apply proper edit distance to complete the search. The key is that q-gram distance is a lower bound for edit distance. Similarly, we could linearly project down a very high-dimensional space (such as 50 by 50 pixel images) to a somewhat lower dimensional space (e.g. by averaging together 2 by 2 pixel blocks) and use that as an approximation (L_2 distance in the projection is a lower bound for L_2 distance in the original space). All of these techniques, of course, rely on special knowledge of the metric space to construct the approximations. However, given the approximations, a general method could be applied.

8 Acknowledgments

I thank Prof. Michael Brin, Prof. Héctor García-Molina, Prof. Frank Olken, Luis Gravano, Edouard Bugnion, and Sandeep Singhal for helpful discussions and for listening to my endless ramblings.

References

[BFR+93] E. Bugnion, S. Fei, T. Roos, P. Widmayer, and F. Widmer. A spatial index for approximate multiple string matching. In *Proc. First South American Workshop on String Processing, Belo Horizonte, Brazil*, pages 43–53, nivio@dcc.ufmg.br, September 1993.

[BK73] W. A. Burkhard and R. M. Keller. Some approaches to best-match file searching. *Communications of the ACM*, 16(4), April 1973.

[FN75] K. Fukunaga and P. M. Narendra. A branch and bound algorithm for computing k-nearest neighbors. *IEEE Trans. Comput.*, C-24:750–753, July 1975.

[FS82] C.D. Feustel and L. G. Shapiro. The nearest neighbor problem in an abstract metric space. *Pattern Recognition Letters*, pages 125–128, December 1982.

[HKR93] D. P. Huttenlocher, G. A. Klanderman, and W. J. Rucklidge. Comparing images using the hausdorff distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(3):850–63, September 1993.

[SW90] Dennis Shasha and Tsong-Li Wang. New techniques for best-match retrieval. *ACM Transactions on Information Systems*, 8(2):140–158, April 1990.

[Uhl91] J. Uhlmann. Satisfying general proximity / similarity queries with metric trees. *Information Processing Letters*, 40(4):175–9, November 1991.

[Ukk92] E. Ukkonen. Approximate string matching with q-grams and maximal matches. *Theoretical Computer Science*, 92(1):191–211, January 1992.

[Yia93] P. N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, pages 311–321, 1993.