Implementing Metric Trees to Satisfy General Proximity/Similarity Queries

Jeffrey K. Uhlmann
Information Technology Division
Naval Research Laboratory
Washington, D.C. 20375-5000

November 23, 1991

Abstract

Efficient metric tree construction and search routines are developed and ANSI C source code with test results are provided. The routines will permit the satisfaction of proximity/similarity queries in arbitrary metric spaces. An efficient, dynamic classification technique is also described.

Efficient search is a central component of many important real-world applications in artificial intelligence, pattern recognition, and pattern classification and correlation [1-4]. In this paper, efficient routines will be developed for satisfying a very general class of search problems. In particular, data structures called metric trees will be described that can permit the fast identification of objects that are close, or similar, to a given query object [5], the only requirement being that the appropriate distance measure represents a metric.

A metric distance function $d(x, y)$ satisfies the following:

(i) $\quad d(x, y) = d(y, x)$

(ii) $\quad 0 < d(x, y) < \infty, \quad x \neq y$

(iii) $\quad d(x, x) = 0$

(iv) $\quad d(x, y) \leq d(x, z) + d(z, y) \qquad$ (triangle inequality)

Most practical distance measures satisfy these conditions.[1] Consider for example $d(x, y)$ giving the minimum number of insertions, deletions, and substitutions of characters necessary to transform a string x into a string y [6,7]. Condition (i) is satisfied simply by

[1] Common distance measures that do violate one of these conditions include those that involve distances between perimeters of sized objects. For example, the distance between the perimeters of two small spheres may be larger than the sum of the distances from each to a large sphere between them, thus violating condition (iv).

reversing the roles of insertion and deletion and of characters used in substitutions. For example, transforming "bats" into "cat" requires the substitution $b \rightarrow c$ and the deletion of the character "s". Transforming "cat" into "bats", conversely, requires the substitution $c \rightarrow b$ and the insertion of the character "s". Conditions (ii) and (iii) are trivially satisfied, and condition (iv) is always satisfied since the transformation of a string x to a string z, followed by a transformation from z to y, clearly must use at least as many edit operations as minimally required to directly transform x into y. The string edit metric has important applications in biology for studying genetic sequences, and in large database management systems for identifying redundant entries that appear different only because of data entry errors. The string edit metric is also important because it does not permit the use of traditional data structures for performing efficient similarity queries.

Ball Decompositions

A ball decomposition is a binary tree structure constructed by recursively partitioning a set of objects according to whether they are inside or outside of a specified ball-type region. More specifically, one of the objects is selected and a radius is identified such that half of the objects are within that radius from the selected object, and half are not. It is useful in practice to identify *two* radii: the smallest radius that encloses half of the objects, called the *inner* radius, and the largest radius that encloses only those objects, called the *outer* radius (See Fig. 1).

A ball decomposition can be constructed by the following procedure:

ConstructBallTree(*BallTree*, *SetOfObjects*)

1. If *SetOfObjects* is empty, then return *BallTree*.
2. If *SetOfObjects* has only one object, q, then:

   Set $BallTree_{\text{object}}$ to q,
   Set $BallTree_{\text{inner}}$ and $BallTree_{\text{outer}}$ to 0,
   Return *BallTree*
3. Select an arbitrary object q from *SetOfObjects*.
4. Set $BallTree_{\text{object}}$ to q.
5. Partition *SetOfObjects* into two equally sized sets, L and G, such that the objects in L have distances from q that are less than (or equal to) those of objects in G.
6. Set $BallTree_{\text{inner}}$ to the distance of the object in L furthest from q.
7. Set $BallTree_{\text{outer}}$ to the distance of the object in G closest to q.
8. Set $BallTree_{\text{left}}$ to ConstructBallTree(*Empty tree*, L).
9. Set $BallTree_{\text{right}}$ to ConstructBallTree(*Empty tree*, G).
10. Return *BallTree*.

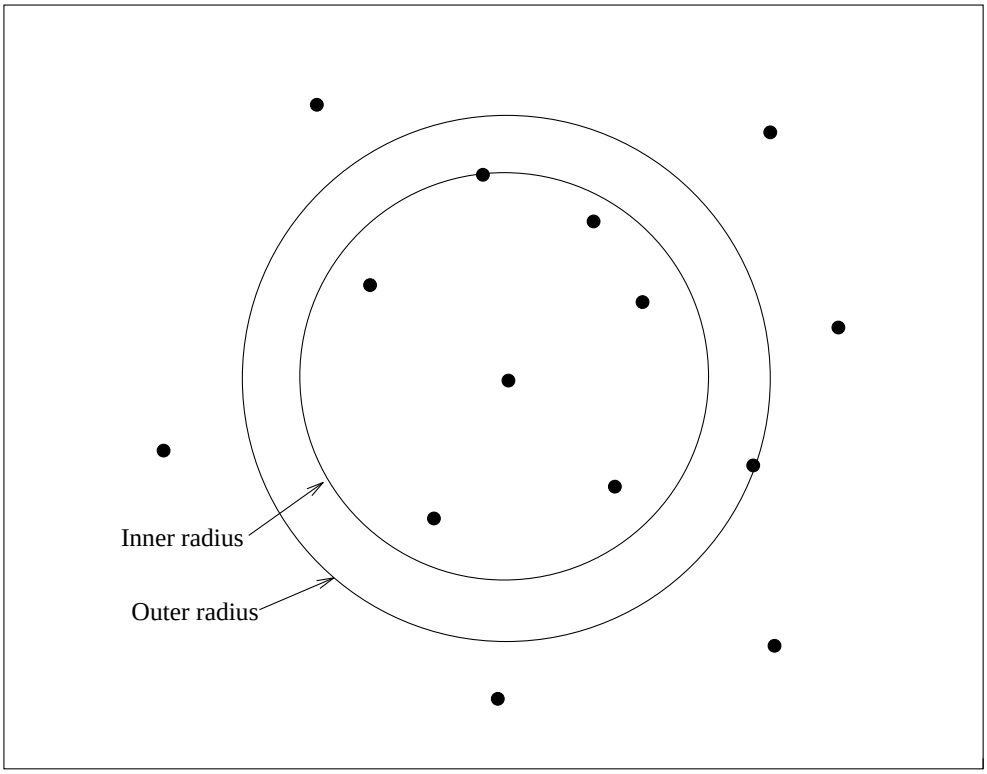

Figure 1: Inner and outer radii of a ball partition

For a set of objects of size N, no step requires more than $O(N)$ computations. Thus, the overall scaling of the construction routine is $O(N \log N)$.

After construction, the following information is stored at each node of the tree:

$BallTree_{\text{object}}$ - the partitioning object

$BallTree_{\text{inner}}$ and $BallTree_{\text{outer}}$ - the inner and outer radii, respectively

$BallTree_{\text{left}}$ - the subtree constructed from the objects within the partition radii

$BallTree_{\text{right}}$ - the subtree constructed from the objects beyond the partition radii

The simplest query that can be satisfied is determining whether a given object is in the tree. This query can be satisfied with only $O(\log N)$ distance calculations as follows:

FindObject(*BallTree*, *Object*)

1. If *BallTree* is an *Empty tree*, return *False*
2. If $BallTree_{\text{object}}$ equals *Object*, return *True*
3. If $d(Object, BallTree_{\text{object}}) \leq BallTree_{\text{inner}}$, return FindObject($BallTree_{\text{left}}$, *Object*);

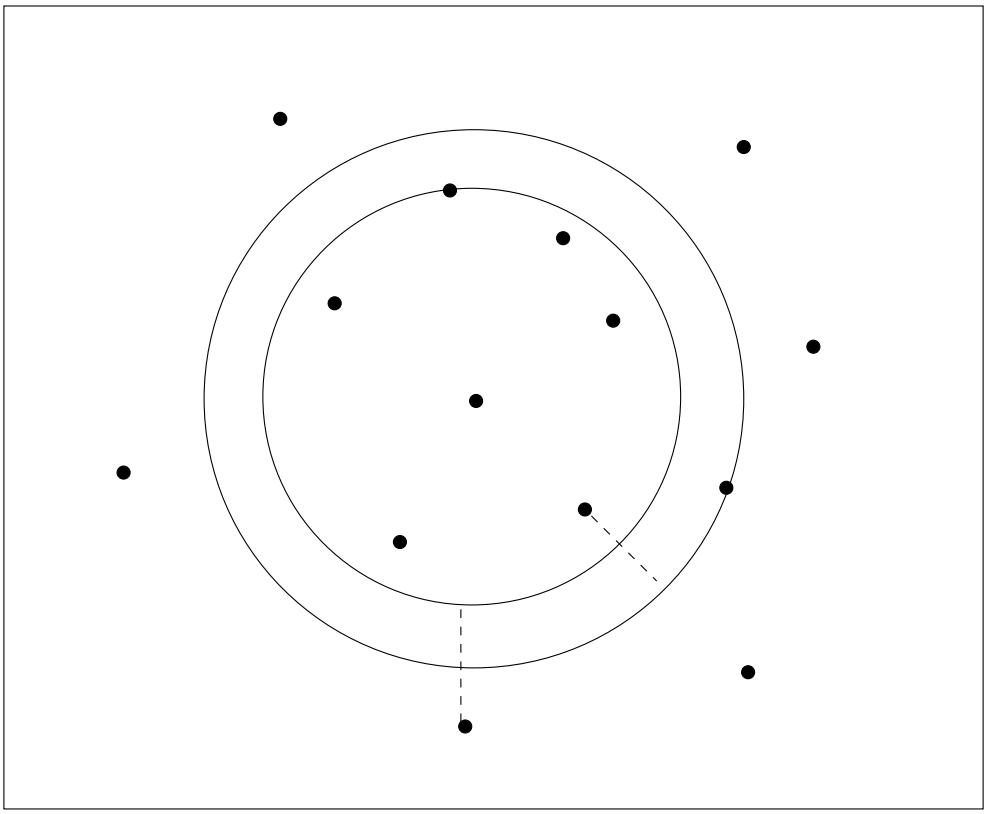

Figure 2: Distances from objects to partitions

4. If $d(Object, BallTree_{\text{object}}) \geq BallTree_{\text{outer}}$, return FindObject($BallTree_{\text{right}}$, $Object$);
5. Return *False*

Note that the procedure stops searching if it finds that *Object* lies between the inner and outer radii at a node. This additional stopping condition provides only trivial performance improvement in this case. For nearest object queries, however, the two radii significantly reduce the amount of searching required.

The difficulty in satisfying nearest object queries is that at some (possibly many) nodes, both subtrees may have to be searched. This is because the given object may be within the inner radius while its nearest neighbor is beyond the outer radius, or vice verse. Searching both subtrees at every node results in the comparison of the given object to every object in the tree. Therefore, some criterion is needed that hopefully can allow many subtrees to be pruned from the search while assuring that the nearest neighbor is always found. *Proximity priority* is such a criterion.

The proximity priority criterion simply associates a cost with each encountered subtree that reflects the minimum distance the closest object in that subtree could possibly be from the given object. For example, if the given object is within the inner radius of a node, the closest object to it in the left subtree could be of zero distance. Thus, the cost associated with the left subtree would be zero. The closest possible object in the right subtree, however, would be one that lies on the outer radius. So the cost associated with the right subtree would be

the distance from the given object to the outer radius (See Fig. 2). The search procedure at each node, then, consists of searching one subtree and storing the cost of searching the other into a priority queue. Whenever a leaf node is reached, the search is continued by using the priority queue to identify the unexamined subtree having least cost.

The following procedure uses the proximity priority criterion to satisfy closest object queries. It assumes that the priority queue initially contains only the tree to be searched (with a cost of zero). The closest object will be assigned to the global variable *ClosestObject*.

FindClosest(*BallTree*, *Object*)

Do while *Priority Queue* is not empty

1. If *BallTree* is an *Empty tree*, set *BallTree* to the subtree having least cost in the *Priority Queue*
2. If the cost associated with *BallTree* is greater than or equal to the distance from *Object* to *ClosestObject*, return
3. If *Object* is closer to $BallTree_{\text{object}}$ than to *ClosestObject*, then set *ClosestObject* to $BallTree_{\text{object}}$
4. If $BallTree_{\text{left}}$ is not an *Empty tree*, then:
   If *Object* is beyond $BallTree_{\text{inner}}$, then the cost in the *Priority Queue* to search the left subtree is the distance from *Object* to $BallTree_{\text{object}}$, minus $BallTree_{\text{inner}}$
   Else the cost to search the left subtree is zero
5. If $BallTree_{\text{right}}$ is not an *Empty tree*, then:
   If *Object* is within $BallTree_{\text{outer}}$, then the cost in the *Priority Queue* to search the right subtree is $BallTree_{\text{outer}}$ minus the distance from *Object* to $BallTree_{\text{object}}$
   Else the cost to search the right subtree is zero

End while

The routine for finding the closest object easily can be enhanced to find the k closest objects. Unfortunately, the number of nodes visited is highly dependent on the distance of the furthest of the k objects from the specified object. Thus, k closest object queries are not generally efficient unless there is information to suggest that there are in fact k objects (even when $k = 1$) close to the specified object. An alternative type of query asks for all objects that are within a given radius of a specified object. This query has the advantage that it puts explicit bounds on the search, unlike closest object queries, and because it can be satisfied without using elaborate data structures such as priority queues. The following simple procedure satisfies radius queries:

SearchRadius(*BallTree*, *Object*, *Radius*)

1. If *BallTree* is an *Empty tree*, return
2. If *Object* is within *Radius* of $BallTree_{\text{object}}$, add $BallTree_{\text{object}}$ to the solution set
3. If $d(Object, BallTree_{\text{object}}) + Radius \geq BallTree_{\text{outer}}$, then SearchRadius($BallTree_{\text{right}}$, *Object*, *Radius*)

4. If $d(Object, BallTree_{\text{object}}) - Radius \leq BallTree_{\text{inner}}$, then SearchRadius($BallTree_{\text{left}}$, *Object*, *Radius*)

(Note that elements from the above two procedures can be combined to find k nearest neighbors within a given radius. This type of query is useful in many types of multi-object correlation problems.)

Ball decompositions are important because they can be used to satisfy a variety of queries for a very general class of objects. The efficiency with which ball decompositions can be applied, however, depends heavily on the metric and on the distribution of the objects. In some applications, speed of retrieval is more important than finding the exact solution set for a given query. For example, being given the second or third closest object in response to a closest object query may be acceptable if the time required to find this approximate match can be bounded to, say, $O(\log N)$. This can be accomplished by placing an upper bound, k, on the number of nodes visited in FindClosest. This amounts to simply changing the line "Do while *Priority Queue* is not empty" to "For $i = 1$ to k while *Priority Queue* is not empty", and adding a line to free the priority queue before returning. Appendix I provides example ANSI C code for the general purpose construction and search of ball decompositions. Appendix II describes an alternative metric tree approach to approximate search. And Appendix III provides test results for ball decompositions on queries involving binary vectors.

References

[1] J. Uhlmann, The complexities of multiple-target tracking, American Scientist, 4/1992.

[2] F. Preparata and M. Shamos, Computational Geometry: An Introduction, (Springer-Verlag, New York, 1985).

[3] H. Samet, The Design and Analysis of Spatial Data Structures, (Addison-Wesley Publishing Co., 1990) Preface.

[4] J. Uhlmann, Adaptive partitioning strategies for ternary tree structures, Pattern Recognition Letters, 9/1991.

[5] J. Uhlmann, Satisfying general proximity/similarity queries with metric trees, Info. Proc. Letters, 11/1991.

[6] D. Sankoff and J. Kruskal (eds.), Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison, (Addison-Wesley, 1983).

[7] M. Atallah and S. McFaddin, Sequence comparison on the connection machine, Purdue University report CSD-TR-853, 1989.

[8] B. Moret and H. Shapiro, Algorithms from P to NP: Vol. I, (Benjamin/Cummings, 1991).

Appendix I

```
typedef struct mtree_type {
    void *Object;
    float InnerRadius, OuterRadius;
    struct mtree_type *Left, *Right;
} Node;


/* Define global name for distance function to
   avoid an extra parameter in recursive ConstructTree */
float (*Distance)(void *Object1, void *Object2);

Node *MakeMetricTree(Node **ObjectArray, int Count,
      float (*ObjectDistanceMeasure)(void *Object1, void *Object2))
{
    Node *ConstructTree(int First, int Last);
    Array = ObjectArray;
    Distance = ObjectDistanceMeasure;
    return(ConstructTree(0, Count));
}


Node *ConstructTree(int First, int Last)
{
    int i, FindMedian(int First, int Last);
    Node *temp;
    int MidIndex, InnerIndex;

    /* Is this a leaf node? */
    if ((Last-First) < 2) {
        if ((Last-First) == 0) return(NULL);
        temp = Array[First];
        temp->Left = temp->Right = NULL;
        return(temp);
    }

    /* Find Inner and Outer Radii */
    for (i=First+1; i<Last; i++)
        Array[i]->InnerRadius =
            (*Distance)(Array[First]->Object, Array[i]->Object);
    MidIndex = FindMedian(First+1, Last);
    Array[First]->OuterRadius = Array[MidIndex]->InnerRadius;
    Array[First]->InnerRadius = 0.0;
    InnerIndex = First;
    for (i=First+2; i<MidIndex; i++)
```

```
        if (Array[i]->InnerRadius >
            Array[InnerIndex]->InnerRadius) InnerIndex = i;
    Array[First]->InnerRadius = Array[InnerIndex]->InnerRadius;

    Array[First]->Left = ConstructTree(First+1, MidIndex);
    Array[First]->Right = ConstructTree(MidIndex, Last);

    return(Array[First]);
}



/* Assumes priority queue functions PQ_DeleteMin, PQ_GetMinKey,
   Free_PQ, and PQ_Insert for priority queues defined as type PQ.
   Since more insertions than deletions will be performed, a
   priority queue with constant time insertion may be preferred.
   (See reference [8] for information on priority queues)    */

void *FindBestMatch(void *Object, Node *Tree,
      float (*Distance)(void *Object1, void *Object2))
{
    Node *Branch;
    float BestDistanceSoFar, DistanceToNode, BranchDistance;
    PQ *NextBranch;
    void PQ_Insert(void *Object, float Key, PQ **PriorityQueue),
        PQ_DeleteMin(PQ **PriorityQueue), Free_PQ(PQ **PriorityQueue),
        *BestObjectSoFar;

    if (Tree == NULL) return(NULL);

    BestObjectSoFar = Tree->Object;
    BestDistanceSoFar = (*Distance)(Object, Tree->Object);

    NextBranch = NULL;
    PQ_Insert(Tree, 0.0, &NextBranch);

    while (NextBranch) {
        BranchDistance = PQ_GetMinKey(NextBranch);

        if (BranchDistance >= BestDistanceSoFar) {
            Free_PQ(NextBranch);
            return(BestObjectSoFar);
        }

        Branch = DeleteMin(&NextBranch);
        DistanceToNode = (*Distance)(Object, Branch->Object);
        if (DistanceToNode < BestDistanceSoFar) {
            BestDistanceSoFar = DistanceToNode;
```

```
            BestObjectSoFar = Branch->Object;
        }
        /* Cost for branching Left */
        if (Branch->Left) {
            if (DistanceToNode > Branch->InnerRadius)
                PQ_Insert(Branch->Left,
                        DistanceToNode - Branch->InnerRadius, &NextBranch);
            else PQ_Insert(Branch->Left, 0.0, &NextBranch);
        }

        /* Cost for branching Right */
        if (Branch->Right) {
            if (DistanceToNode < Branch->OuterRadius)
                PQ_Insert(Branch->Right,
                        Branch->OuterRadius - DistanceToNode, &NextBranch);
            else PQ_Insert(Branch->Right, 0.0, &NextBranch);
        }
    }
    return(BestObjectSoFar);
}


int FindMedian(int First, int Last)
{
    int Partition(int First, int Last, int Pivot);
    int Mid, Pivot, k;

    Mid = (First + Last) / 2;
    do {
        k = Partition(First, Last, Mid);
        if (k < Mid) First = k + 1;
        else if (k > Mid) Last = k;
    } while (k != Mid);
    return(Mid);
}


Partition(int First, int Last, int Pivot)
{
    register int i, j;
    Node *temp;
    float Radius;

    if ((Last-First) == 1) return(First);
    Last--;
    Radius = Array[Pivot]->InnerRadius;
    temp = Array[First];
    Array[First] = Array[Pivot];
```

```
    Array[Pivot] = temp;
    i = First;
    j = Last;
    if (Array[Last]->InnerRadius > Radius) {
        temp = Array[First];
        Array[First] = Array[Last];
        Array[Last] = temp;
    }
    while (i < j) {
        temp = Array[i];
        Array[i]  = Array[j];
        Array[j] = temp;
        i++;  j--;
        while (Array[i]->InnerRadius < Radius) i++;
        while (Array[j]->InnerRadius > Radius) j--;
    }
    if (Array[First]->InnerRadius == Radius) {
        temp = Array[First];
        Array[First] = Array[j];
        Array[j] = temp;
    }
    else {
        j++;
        temp = Array[Last];
        Array[Last] = Array[j];
        Array[j] = temp;
    }
    return(j);
}
```

Appendix II

Finding an exact match of a given object in a ball decomposition can be satisfied in strict $O(\log N)$ time because at each node in the tree it is known in which subtree the exact match must lie. In the case of best match queries, however, it is possible that the given object and its best match will fall on different sides of a partition, and the curvature of ball partitions can actually increase the likelihood of this event occuring. An alternative type of metric tree can minimize this problem.

Definition: A *generalized hyperplane* (GH) is defined by two objects p_1 and p_2, $p_1 \neq p_2$, and consists of the set of objects q satisfying $d(q, p_1) = d(q, p_2)$. An object x is said to lie on the p_1-side of the plane if $d(x, p_1) < d(x, p_2)$.

GH decompositions have several advantages over ball decompositions. For example, ball decompositions are strongly static data structures while GH decompositions are not. In particular, a ball partition consists of a center point and a radius that is determined by examining each object in the dataset and identifying the median distance from that center point. Without knowledge about the spatial distribution extent of the objects to be processed, there can be no strategy for dynamically selecting radii that can be expected to produce an approximately balanced data structure.[2] A heuristic sampling argument, however, suggests that GH partitions usually can be expected to produce approximately balanced decompositions because they are determined by randomly sampled objects (the pairs of defining objects) that they partition into equally sized subsets (each partition separates its two defining objects). Calculating the expected deviation from perfect balance, of course, requires additional distribution and metric information.

The dynamic insertion of a object p into a GH-based tree $\mathcal{H}$ can be effected by the function Insert$(\mathcal{H}, p)$ defined as follows:

1. If $\mathcal{H}$ is empty, then create and return a node $\mathcal{H}$ with $\mathcal{H}_x = p$ and $\mathcal{H}_y, \mathcal{H}_{\text{left}}$, and $\mathcal{H}_{\text{right}} = nil$, where $\mathcal{H}_x$ and $\mathcal{H}_y$ are the two objects defining the partition and $\mathcal{H}_{\text{left}}$ and $\mathcal{H}_{\text{right}}$ are the two subtrees.
2. If $\mathcal{H}_y = nil$, then return $\mathcal{H}$ with $\mathcal{H}_y = p$.
3. If $d(p, \mathcal{H}_x) \leq d(p, \mathcal{H}_y)$, then return $\mathcal{H}$ with $\mathcal{H}_{\text{left}} = \text{Insert}(\mathcal{H}_{\text{left}}, p)$.
4. Return $\mathcal{H}$ with $\mathcal{H}_{\text{left}} = \text{Insert}(\mathcal{H}_{\text{right}}, p)$.

The advantages of GH over ball decompositions in the dynamic case are gained at a cost. Without knowing that the distance measure defines a linear space, for example, it is not generally possible to translate generalized hyperplanes to assure balanced decompositions even in the static case. (If the application does involve a distance measure defining a linear vector space, virtually every technique described in this report for using ball decompositions can be applied with better results to GH decompositions.) It is also not generally possible

[2] However, dynamic techniques have been developed that yield good *amortized* performance for insertions and deletions in general tree structures. These approaches perform no balancing until a subtree becomes sufficiently unbalanced, whereupon it is completely reconstructed by using the static construction algorithm [4].

to determine the minimum distance between an arbitrary object and a GH partition. For example, given only a blackbox distance function, two objects defining a partition plane, and a ball of radius r about an object x, it is not possible to determine whether the ball about x intersects the partition plane. Thus, GH decompositions are less flexible than ball decompositions.

Despite their limitations, GH decompositions can be used to satisfy one class of approximate best match queries in any metric space. Specifically, the closest object encountered during the search prescribed for an exact match can be considered an approximate best match. This approach has applications to pattern classification. For example, given a set of training patterns, with their associated classifications, a metric classification tree can be constructed as follows: *For each training pattern, determine if its classification is the same as its approximate best match in the tree; if not, then insert the pattern into the tree. Repeat over the training set until no new patterns are inserted.* The constructed tree will correctly classify each of the patterns in the training set, but without necessarily having to store all of the patterns. The tree can then be used to predict the classification of new patterns. The greater the correlation between proximity and classification, the better the predictions will be. Of course, if a misclassification is identified, the tree can be dynamically updated by inserting the misclassified pattern with its correct classification.

The following is example ANSI C code for inserting an object into a GH tree, and code for finding approximate best-matches in a GH tree:

```
typedef struct tree_node {
    void *p1, *p2;
    struct tree_node *left, *right;
} Node;


Node *Insert(Node *Tree, void *Object, float (*Distance)(void *, void *))
{
    if (Tree == NULL) {
        if ((Tree = (Node *) malloc(sizeof(Node))) == NULL) {
            printf("Insufficient storage in Insert()\n");
            exit(1);
        }
        Tree->p1 = Object;  Tree->p2 = NULL;
        Tree->left = Tree->right = NULL;
    }
    else if (Tree->p2 == NULL) Tree->p2 = Object;
    else if ((*Distance)(Tree->p1, Object) < (*Distance)(Tree->p2, Object))
        Tree->left = Insert(Tree->left, Object, Distance);
    else Tree->right = Insert(Tree->right, Object, Distance);
    return(Tree);
}
```

```
void *Closest_Object;
float Distance_to_Closest;

void *Search(Node *Tree, void *Object, float (*Distance)(void *, void *))
{
    float Distance_to_p1, Distance_to_p2;

    if (Tree == NULL) return(Closest_Object);
    NodesVisited++;

    Distance_to_p1 = (*Distance)(Tree->p1, Object);
    if (Distance_to_p1 < Distance_to_Closest) {
        Closest_Object = Tree->p1;
        Distance_to_Closest = Distance_to_p1;
    }
    if (Tree->p2 == NULL) return(Closest_Object);
    NodesVisited++;

    Distance_to_p2 = (*Distance)(Tree->p2, Object);
    if (Distance_to_p2 < Distance_to_Closest) {
        Closest_Object = Tree->p2;
        Distance_to_Closest = Distance_to_p2;
    }
    if (Distance_to_p1 <= Distance_to_p2)
        return(Search(Tree->left, Object, Distance));
    if (Distance_to_p2 <= Distance_to_p1)
        return(Search(Tree->right, Object, Distance));
}


/* Finds approximate best match to Object in Tree */
void *Match(Node *Tree, void *Object, float (*Distance)(void *, void *))
{
    Closest_Object = NULL;
    Distance_to_Closest = INFINITY;

    return(Search(Tree, Object, Distance));
}
```

Appendix III

Many data structures have been developed to satisfy certain classes of proximity queries for vectors having d arbitrary real-valued elements. Most of these data structures are based on the paradigm of recursive hyperplane decomposition. The kd tree, for example, is constructed by recursively selecting one of the coordinates and partitioning the dataset into the subset of vectors whose values for the chosen coordinate are less than the median value and the subset whose values for the coordinate are not less than the median. The methods for searching the tree are then closely analogous to those used to search ordinary single-dimension binary trees. This data structure requires storage linear in the size of the dataset and usually displays good query-time performance when the number of dimensions is small. When the number of dimensions exceeds $O(\log n)$, assuming the tree is balanced, no sequence of partitions can discriminate on every coordinate. In other words, a search of the tree can only determine proximity based on a subset of the coordinates.

Data structures such as the kd tree also may be inadequate for efficiently storing discrete-valued vectors in which several vectors have the same value for one or more of the coordinates. If the median value for a coordinate is not unique, then a partition on that coordinate will produce an unbalanced tree. In the case of binary vectors, a partition on a single coordinate will maintain balance only if half of the objects have the same value for that coordinate. This condition is highly data dependent and does not hold for most practical problems. (However, judicious selection of partitioning coordinates often can avoid such difficulties.)

To demonstrate the performance advantage of the metric tree over brute force approaches, test results are presented of constrained best-match (in Hamming distance) queries on datasets of uniformly distributed binary vectors. Binary vectors are considered for two reasons: (1) binary vectors present difficulties, as discussed earlier, for alternative data structures. And (2), many parameterizations of situations simply define sets of binary *true/false* conditions. The results, however, fairly typify the performance that can be expected for general vectors. Each test shows the effect of one of the critical variables on the number of distance calculations computed in determining best matches. In the first test, the dataset size is varied from $1K$ to $16K$, $K = 1024$. The number of dimensions is fixed at $1K$ and each query vector is generated by corrupting five bits of one of the vectors in the dataset. In other words, each best match is of distance five from the query vector.

The results displayed in Fig. 3 reveal that the the number of distance calculations does not increase proportionally with an increase in the size of the dataset. In particular, a doubling of the dataset size consistently produces an increase of only 28% in the number of required distance calculations. Since the number of binary coordinates is fixed, this suggests that the number of distance calculations has scaling that is sublinear with density, at least for small search radii. Clearly, the number of distance calculations approaches $\log N$ as the search radius approaches zero. (Unfortunately, a satisfactory average-case complexity analysis in terms of the critical variables has not been performed.)

In the second test, the distance between the query vector and its best match in a $1K$

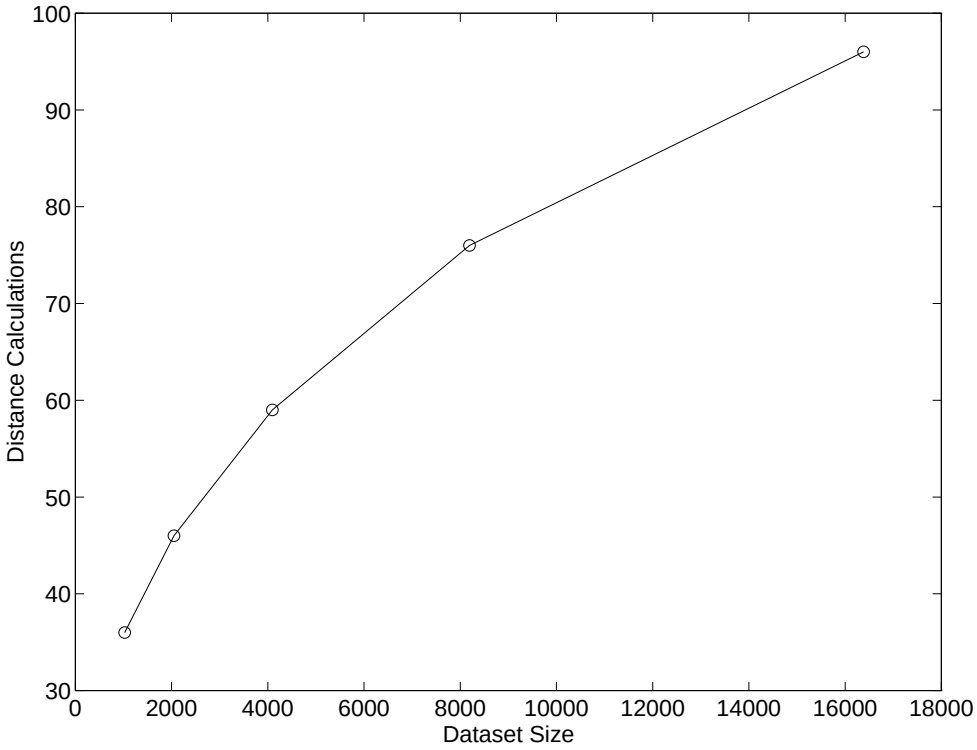

Figure 3: Effect of dataset size

dataset is varied from zero to sixteen. The results demonstrate that the number of distance calculations is highly sensitive to this variable. Additional tests, however, reveal that even in the case of an unconstrained search, a small reduction in the total number of distance calculations can compensate for the increased overhead involved in searching the tree. For applications in which distance/similarity calculations are far more computationally expensive, this suggests that substantial improvements in performance over brute force should be expected.

In the third test, the number of dimensions, or vector length, is varied from $1K$ to $16K$ for a $1K$ dataset. This variable is important because data structures such as the kd tree would only be able to discriminate on a very small subset of the coordinates. In other words, the performance of the kd tree is not improved (and usually degrades) as the amount of information, i.e., dimensionality of the measurements, increases. The results in Fig. 5 demonstrate that the performance of the metric tree improves, in terms of number of distance calculations, as the amount of information on which it can discriminate increases. This improved performance is due to the decreased density, and therefore increased average separation, of the vectors in the dataset.

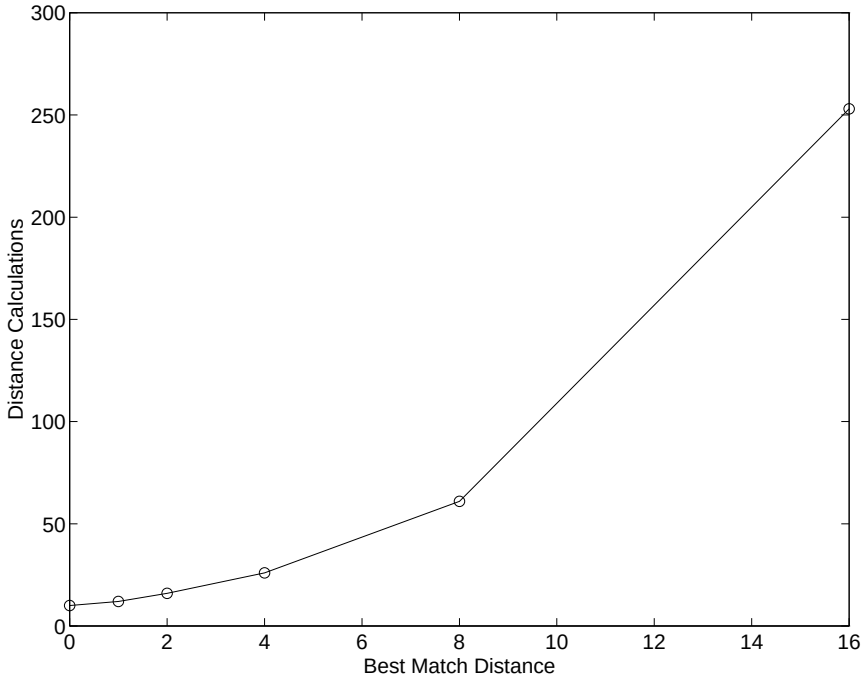

Figure 4: Effect of distance between query vector and its best match

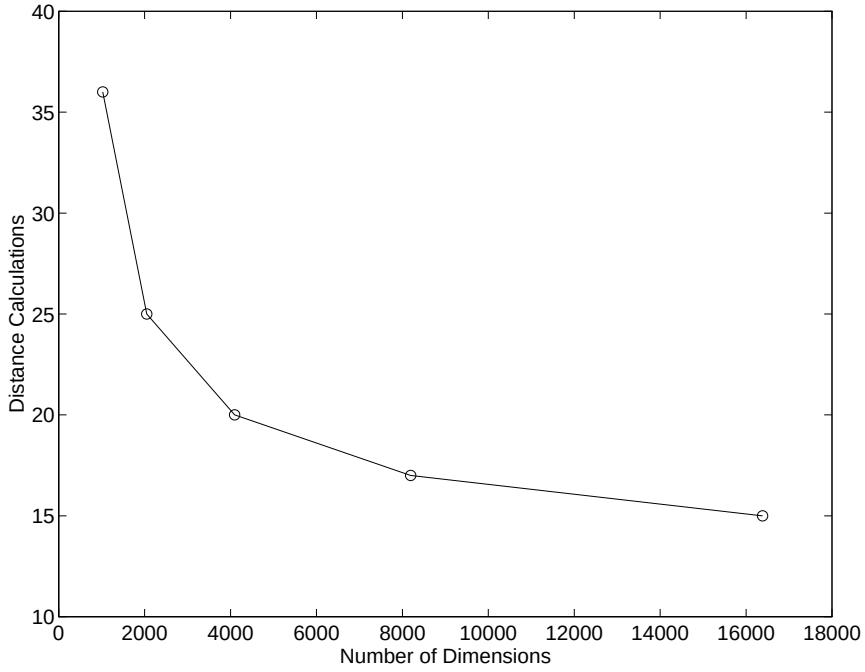

Figure 5: Effect of vector length (dimensionality)