Analysis of Search Algorithms and Tree Structures for Proximity Search in Metric Spaces

by

Neha Singh

Undergraduate Honors Thesis (Fall 2002)

Under the supervision of:

Dr. Daniel P Miranker

Department of Computer Sciences

The University of Texas at Austin

Abstract

Proximity search in metric spaces involves searching the elements of a set that are close to a specified query point when the data elements form a metric space. The triangle inequality is a fundamental property of metric spaces and can be utilized in various ways to prune the metric search space. There are various frameworks under which metric spaces have been organized and the algorithms used to perform proximity queries are dependent on how the metric space tree has been structured. We present a classification of the search strategies based on triangle inequalities and the metric tree indexing algorithms. Algorithms are presented for various combinations of these strategies which result in different trade-offs of the time and space required for the search. Experimental analysis of these algorithms is performed in the context of the biological database management system called MoBIoS (Molecular Biological Information System) that we are developing.

1. Introduction

The goal of the MoBIoS project is to develop a new specialized database management system for biological databases and a database query language that embodies the semantics of genomic and proteomic data. The storage manager of this database management system is based on metric space indexing techniques as the biological information to be stored in it can best be modeled by a metric space. The contents of the database are biological sequences, such as protein, DNA and RNA sequences. They have to be stored in an efficient and structured manner in the database to facilitate the execution of various kinds of queries performed on the database by biologists.

Any search performed on the database involves returning all sequences that are close to a given sequence. The closeness is measured in terms of the global alignment of sequences which is based on simple edit distance that forms a metric (Levenshtein distance) [9]. A radius is specified and any sequence whose distance from the query sequence is less than or equal to the radius is part of the result of performing the query. In effect, the query sequence forms the center of a sphere of the given radius and any sequence falling within the sphere has to be returned.

In general, the above problem can be abstracted as a proximity search query in a metric space. The database search tree is therefore comprised of nodes that form a metric space. The distance function d between each pair of nodes is well-defined, and satisfies the following properties for any three points x, y and z in the metric space:

a.) $d(x,y) >= 0$, $d(x,y) = 0$ iff $x = y$ *(Positivity)*

b.) $d(x,y) = d(y,x)$ *(Symmetry)*

c.) $d(x,z) <= d(x,y) + d(y,z)$ *(Triangle inequality1)*

The three properties above comprise the definition of a metric space [5, 6]. A fourth property can be derived from the triangle inequality: $d(y,z) > | d(x,y) - d(x,z) |$ *(Triangle Inequality 2).*

A query consists of a point Q and a search radius R. It returns all the objects X in the database that lie within a distance R of the query point Q. So the query results satisfy the following inequality:

$$d(X, Q) <= R$$

The distance between the query point and each of the nodes is not initially known. The distance computation, which involves calculating the edit distance between two sequences, is expensive in terms of time. So, one of the main aims is to answer the proximity query with as few distance calculations as possible. This is achieved by exploiting the triangle inequalities (properties c. and d. above) to eliminate or include spheres in the answer set, without actually computing the distances for most of them. The minimization of distance computations by using the triangle inequalities and their variations impose requirements of certain properties on the database tree, which engender different structures for the nodes of the tree. Combinations of a number of different algorithms for search and structures of a database node have been analyzed and presented in this paper. Each of these combinations presents a different trade-off on the space and time efficiency of a search. Another important goal of the search algorithms is to minimize the number of disk input/output operations that are performed for accessing the nodes of the database tree. Fewer the number of nodes that are visited, fewer will be the IO operations.

There are a number of factors that have a bearing on the performance of the proximity search in the metric-space based database tree. The structure of the tree and its contents is one of the most important ones. Efficiency of the search is heavily dependent on the accuracy of the clustering algorithm that is used to group data from the metric space into clusters of closely positioned points. If the physical clusters from the metric space are also grouped together in the branches of the tree, the search will be able to stay in a narrow area and hence perform better. On the other hand, if the physical distances between points do not correspond closely to the grouping of points in the tree, the search suffers because of having to traverse multiple branches, each of which return a small portion of the resultant set. The quality of clusters is therefore one of the biggest determinants of the performance of the search. Also, the structure of the node of a tree, which includes the information that is stored in the node, affects the search because different search algorithms need to be applied to different node structures, as they need different type of information from the tree to execute. A number of classes of searching algorithms are present, such as hyper-plane search methods and vantage point methods. Within these, the algorithms can be varied to use different elimination rules to prune the branches of the tree during search. The choice of the algorithm is interrelated to the structure of the tree and a node in the tree. Different search algorithms have different time and space requirements. Elimination rules can be used in various combinations to create different effects. The search is also guided by the branching factor, which is an attribute of the tree. An optimum average branching factor is dependent on the page size of the database. From all the above factors, the ones that are being analyzed in this paper are the search algorithms and the elimination rules.

There are a number of indicators of the performance of a proximity search algorithm. The foremost among these is the time required to complete the search, which is determined in part by the number of distance calculations that need to be performed to complete the search. The space requirement of the search is related to the size of the tree nodes that were used in it. Another important statistic that can be used to gauge the quality of the search is the fraction of the total number of nodes that were traversed or *visited* as the search progressed down the tree. A high fraction is equivalent to a high number of disk input/output operations, as every node that is examined has to be read from the disk, which does not speak well for the performance of the search.

For the storage manager of MoBIoS, two kinds of tree construction and search mechanisms have been tried: the Generalized Hyper-Plane method and the Vantage Point method. For the hyper-plane method, proximity search algorithms have been developed and implemented for three combinations of the various versions of the triangle inequality. After bulkloading the tree with sequences, extensive testing of the proximity search was performed with each of the algorithms. Statistics were collected during these searches that included all the indicators of performance that were described above. These results were then compared and analyzed.

The paper is organized as follows. In section 2, a method of classification of the search strategies that was developed in order to provide a flexible framework to try different algorithms, is presented. In section 3, the implementation of this framework for the MoBIoS storage manager is introduced. This section details how the various tree nodes from the different classifications have been organized. Also presented is the implementation of the structure of the search mechanism. Section 4 outlines the actual search algorithms based on the triangle inequalities that were developed to perform proximity search on the different types of trees. Section 5 describes the testing framework that was set up for the subsequent experiments and the data that was collected. It then presents an analysis of the results of the experiment runs. Finally section 6 presents the conclusion and future work.

2. Classification of Search Strategies

A classification scheme can be applied to the strategies that are used to perform proximity search on a metric-space based database tree. This classification helps to separate the different parameters of the search, so that a framework can be developed where each of these parameters can be implemented and studied independently. The search strategies that can be applied to metric databases can be classified according to the following criteria:

a.) The version of the triangle inequality that is used for elimination of sub-trees to be searched, in other words, for pruning the tree.

b.) The indexing algorithm on which the search tree is based, such as generalized hyper plane and vantage point, and the search method that will be used.

Variations of the Triangle Inequality

The triangle inequalities are based on a fundamental property of metric spaces. They can be used to prune out a particular child of a node, which causes the sub-tree with that child as the root to be excluded from the search. The objective of using these inequalities instead of using a straight comparison of the distances is to reduce the number of distance calculations between the query and the nodes. This leads to better performance as distance calculations are expensive. The trade-off with the fewer number of distance computations is that these inequalities require certain extra information to be stored in the node of the search tree. The values that are needed in the inequalities can be computed at the time of the construction of the tree and stored in the nodes, so the search time is not affected, but extra disk space is used up as the size of each node is now larger.

The basic data that is needed in the node is the center of the node, its radius and pointers to its children. The following are the different versions of the triangle inequality that can be used individually or in combination with each other as the elimination rules, along with the extra information that a node will need to have in each case:

[*P* represents the parent node; *Q* represents the query center; *C*, *C1*, *C2* represent the centers of the children of the parent node; *r (X)* represents the radius of node *X*; *d (A,B)* represents the distance between the centers of nodes *A* and *B*.]

1. If ***d (C, Q) > r (Q) + r (C)**,* then eliminate *C.*

This inequality uses the distance from the center of a child to the query center to eliminate the child. The radius of each child has to be stored in the parent node, to eliminate the cost of accessing the child node each time to obtain the radius. If there are *n* children for a node, then this rule involves performing *n* distance calculations, between the query center and each child, for every node that is visited.

- Space requirement per node: O (n)
- Search time requirement per node: O (n)

2. If **| *d (P, Q) - d (P,C)* | > *r (Q) + r (C)***, then eliminate *C*.

This inequality uses the distance between the parent and the query, and the distance between the parent and the child to eliminate the child. It requires distances between the parent and each child to be stored in the parent. These distances will have to be computed during the time of the creation of the tree. Also the radius of each child has to be present in the parent node, otherwise the child will have to be accessed for getting the radius. This method entails computing the distance between the parent and the query point during search time, so there is just one distance calculation per every node that is visited.

- Space requirement per node: O (n)
- Search time requirement per node: O (1)

3. If **| *d (C1, Q) - d (C1, C2)* | > *r (Q) + r (C2)***, then eliminate *C2*.

This inequality uses the distance between two children, and the distance between one of those children and the query to eliminate the other child. It involves storing a table of distances between all pairs of children in the parent node, which will be computed at creation time. Also, the radii of the children have to be stored in the parent. There will be approximately *n* more distance calculations between each child and the query point.

- Space requirement per node: O (n^2)
- Search time requirement per node: O (n)

Classification based on Tree Indexing Algorithms

The algorithm that is used for creating the indexed tree structure is an intrinsic determinant of the search strategy that is to be applied because each algorithm results in a structure with certain unique properties that can be exploited in conducting the proximity search. These properties both place constraints on the search as well as aid the search in other situations. The triangle inequalities described above can be used in combination with these structures to get distinct search algorithms with varying levels of performance in terms of time and space.

The two algorithms under investigation for constructing the metric tree structure are:

- the hyper plane method: GHT
- the vantage point method: VPT

1. GHT (Generalized Hyper plane Tree)

A GHT is based on the hyperplane principle [11, 12]. For its construction, 2 centers c_1 and c_2 are selected from the data set. All other elements are divided into the sub-trees of these centers. Each element goes into the sub-tree of the center that it is closest to. This procedure is then recursively applied to each sub-tree to get the GHT structure. The radius of each node is required to be stored in it. The performance of a GHT depends on the choices of nodes that have been selected as centers. It has been found that the minimization of the covering radius of each subtree is the best criterion for the selection of centers. The GHT algorithm can also be generalized to include *n* centers instead of 2.

The algorithm uses the hyperplanes between the centers as the pruning criterion during search time. The left subtree is entered if $d (q, c1) - r < d (q, c2) + r$ and the right subtree if $d (q, c2) - r \leq d (q, c1) + r$. So in general, there are *n* distance calculations required between the query point and each child. Searching a GHT using these triangle inequalities has some characteristics, one of which is that it is possible to enter multiple sub-trees which is desirable for the purposes of a proximity search because the result is not unique, but rather a set of data points.

2. VPT (Vantage Point Tree)

The vantage point tree was introduced as a tree data structure designed for continuous data functions [7, 13]. The basic principle of a VPT is that a node in this tree employs distance from a selected vantage point to divide the space. For the construction, two vantage points v1 and v2 are selected. The rest of the elements are subsequently mapped to a 2-dimensional plane whose axes represent the distances to v1 and v2. Then the medians are computed for both the axes and the points are clustered into rectangles in the 2-D space with the medians as bounds. This results in a structure with concentric spheres in multi-dimensional space. Again the structure can be generalized to an m-ary tree by using $m - 1$ uniform percentiles instead of just the median [1, 2].

The search queries are also mapped into the 2-D space. All rectangles which the query intersects are searched to find the required answers. In other words, if $d\,(q, p) - r \leq M$, we enter into the left subtree and if $d\,(q, p) + r > M$, we enter into the right subtree. Both subtrees can be entered.

3. Interface and Data Structure Definitions for Proximity Search

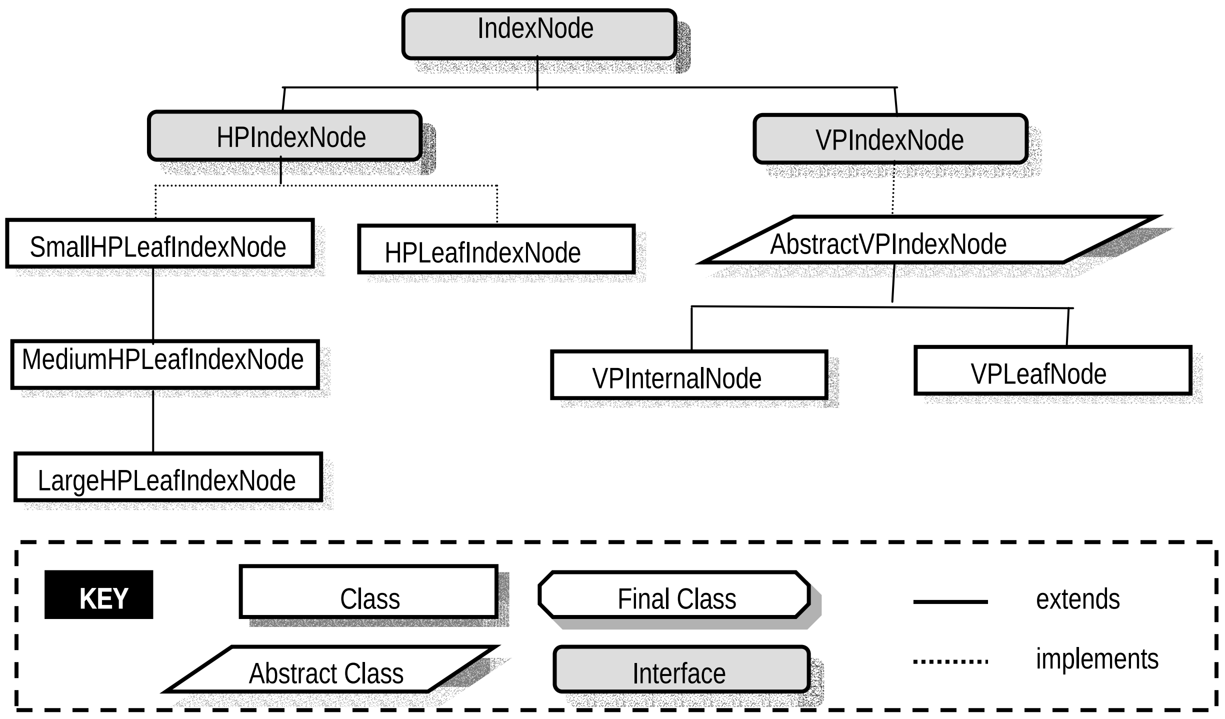
For the purpose of implementing various search algorithms based on combinations of strategies from the classification presented above, a flexible and extensible framework had to be developed in the storage manager of MoBIoS. The aim was to separate the following three components: algorithms used for clustering of the data and construction of the tree; the structure of the nodes of the tree; and the algorithms used for searching the tree. Developing these modules independently from each other facilitates the use of any combination of them, to give a wider range of possibilities to experiment with. Also, it aids in the extension of this framework to support more types of trees or more types of nodes which may be defined later and will easily fit in with the rest of the structure, with minimum modification of the existing body of code.

The objective of this paper is to explore the two latter components: the structure of nodes of the tree and the search algorithms. The method that was used for construction of the tree for the experiment results presented here is a hierarchical bulkloading algorithm that alternates between top down and bottom up clustering to initialize the index [10]. The implementation of the MoBIoS storage manager is done in Java.

Interface for the Structure of Tree Nodes

The nodes that constitute the database tree can be of two types. Internal nodes are present in the interior of the tree and contain pointers to other nodes of the tree. Their children are other internal nodes or leaf nodes. The real data is present in the bottom level of the tree. The leaf nodes are the nodes at the lowest level that contain pointers to the data. Each leaf node can contain references to multiple data points.

There are two types of tree structures that are being implemented: the generalized hyper-plane tree and the vantage-point tree. Hence there are two broad types of node structures: the HP nodes and the VP nodes. Each of these types has internal and leaf nodes. Within the HP class of nodes, three different types of internal nodes have been implemented that use varying combinations of the triangle inequalities. The following is a diagram representing the relationships of the interfaces and classes that represent the data structure nodes:



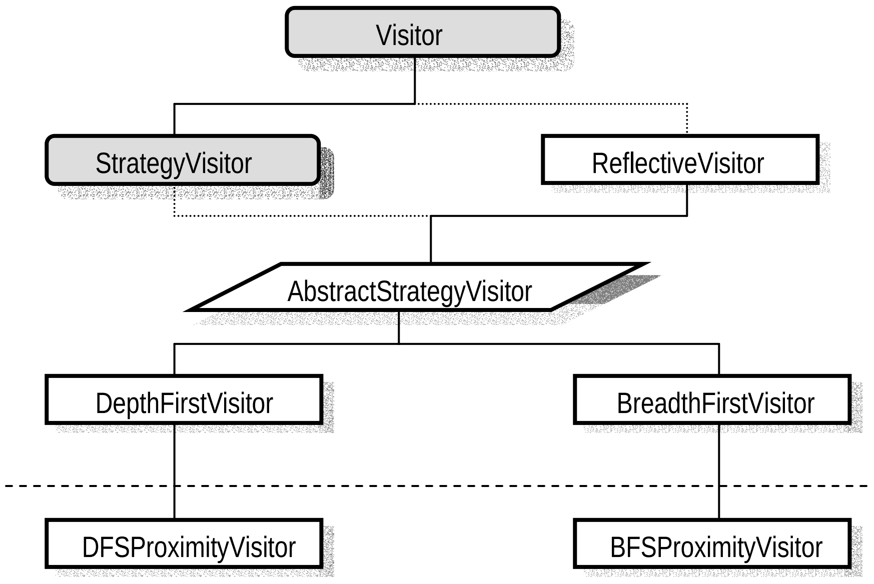
IndexNode is the base interface for representing both hyper plane and vantage point nodes. It has methods to get a child node given the index and to return the number of children of a node. *HPIndexNode* inherits from *IndexNode*. It is an interface to represent the common functionality of internal as well as leaf nodes of a GHT, and has methods to get the radius and center of a node. The *HPLeafIndexNode* implements *HPIndexNode* and represents the leaf nodes of a hyper plane tree. Its data members include the center, the radius and an array of references to its children, which are actual data points, an array containing the centers of the children and an array of distances from the parent to each child. It has methods to access all the data members. The number of children of each node is determined by the desired page size and occupancy. The three types of internal nodes are called *SmallHPInternalIndexNode*, *MediumHPInternalIndexNode* and *LargeHPInternalIndexNode*, according to their size which depends on the amount of data they store. The following table summarizes the properties of the HP internal nodes :

Type of Node	Inequalities used for Pruning	Elimination Equations	Data Members
Small	2	| d (P,Q) - d (P,C) | > r (Q) + r (C)	Center, radius, array of references to children, array of radii of the children, array of distances between the parent and each child
Medium	1,2	d (C,Q) > r (Q) + r (C) | d (P,Q) - d (P,C) | > r (Q) + r (C)	Inherits all data members of *SmallHPInternalIndexNode*, also has an array that stores the centers of the children
Large	1,2,3	d (C,Q) > r (Q) + r (C) | d (P,Q) - d (P,C) | > r (Q) + r (C) | d (C1,Q) - d (C1,C2) | > r (Q) + r (C2)	Inherits all data members of *MediumHPInternalIndexNode*, also has an array that stores the distances between each pair of children

The other branch of the index nodes is the one that implements the vantage point structure. V*PIndexNode* inherits from *IndexNode* and is the interface to represent the common properties of internal and leaf nodes of a VPT. It has methods to get the size of the sub-tree below the current node, to get the number of vantage points and to get the vantage point given an index. The class *AbstractVPIndexNode* implements *VPIndexNode*. It is an abstract class to implement the common functionality of internal and leaf nodes of a VPT. Its data members include an array of vantage points (2 in this case) and the tree size. It also has methods to access the data members. A *VPLeafNode* inherits from *AbstractVPIndexNode* and provides the implementation of the leaf node of a VPT. Its data members consist of an array of distances from the first vantage point to each child, array of distances from the second vantage point to each child and an array of references to children, which are actual data objects for a leaf node. Finally, the *VPInternalNode* also inherits from *AbstractVPIndexNode* and provides implementation of the internal node of a VPT. Its data members are arrays of lower and upper bounds of distances of children clusters to the first and second vantage points and an array of references to children, which are other internal or leaf *VPIndexNodes*.

Interfaces for Implementation of Search

The proximity search mechanism of MoBIoS has been implemented using the *Visitor* pattern. The purpose of the *Visitor* pattern is to encapsulate an operation that has to be performed on the elements of a data structure. It allows the flexibility of changing the operation being performed on the structure without the need of changing the classes of the elements that are being operated on [8]. Using this pattern for the search decouples the classes for the node structures from the search algorithms that are used upon them.



The diagram on the side depicts the interfaces and classes being used for the search. All the classes above the dashed line are primarily used for implementing the *visitor* mechanism, where as the classes below the dashed line are the ones that actually implement the search algorithms on the index nodes. *Visitor* provides a minimal interface for classes implementing the *visitor* pattern. An implementation needs to provide *visit* methods, each with a parameter whose type matches the set of objects to visit. The dispatch technique to select the right visit method for a given object is implemented by using java reflection in *ReflectiveVisitor*. In addition to this dispatch technique, *StrategyVisitor* provides the interface for a strategy to walk data structures, which is implemented by a set of methods to access and retrieve the objects to visit. Concrete implementations of the strategy are provided in *DepthFirstVisitor* and *BreadthFirstVisitor*. Using these classes will result in the elements being visited in an order that they would be returned by a depth-first search and a breadth-first search respectively.

The *ProximityVisitor* classes need to be initialized with the search parameters such as the query center and the search radius. *DFSProximityVisitor* traverses the tree in depth-first order and *BFSProximityVisitor* visits the nodes in a breadth-first order. Both these classes contain *visit* methods for each type of index node that has been implemented. So when traversing a tree constructed with a *MediumHPInternalIndexNode*, for example, the *visit* method for this type of node is invoked. The *visit* method contains the algorithm for searching the node of the specified type which includes pruning it using the combination of triangle inequalities that was outlined earlier. The advantage of using the *visitor* scheme for encapsulating the search is that when new types of node structures are added to the system, the only modification that is needed is addition of new *visit* methods in the *ProximityVisitor* classes.

4. Algorithms for Proximity Search

GHT Proximity Search Algorithms

Three proximity search algorithms have been implemented for a generalized hyperplane tree for the small, medium and large HP internal index nodes.

The first algorithm for the small nodes uses just the second version of the triangle inequality. The node contains its own radius and center and an array of references to its children. Other information that is available to the node that is used for pruning the search is an array containing the radii of the children and an array containing the distance from the node to each of its children. The children radii are stored in the parent so that the child does not have to be accessed for fetching its radius because that would lead to an extra disk input/output operation. The parent-children distances are pre-computed at creation time and stored to save time during the search. The space requirement for the nodes is O (*n*) and complexity of the pruning algorithm for a node is O (*n*) if *n* is the number of children of the node.

Algorithm 1:

```
  visit_small_HP_internal (N: small_HP_internal_node, q: query center, r: search radius)
  {     parent_query_distance = d (N, q)
        if (parent_query_distance > r + N.radius)
                prune the current node N

        for each child i of N
        {       if ( | parent_query_distance - N.parent_child_distance(i) | ≤ r + N.child_radius(i) )
                        add child i to list to be searched
                else
                        prune child i
        }
  }
```

The second algorithm for medium nodes uses the first and the second inequalities together. The node contains its center, radius, references to its children, array of children's radii, array of distances from the parent to the child, and also an array of centers of the children. The centers of the children are stored in the parent so that the children do not have to be accessed in order to get their centers, which are required for this algorithm in the parent. The space requirement for the medium node is O (*n*) and the time complexity of the algorithm is O (*n*).

Algorithm 2:

```
  visit_medium_HP_internal (N: medium_HP_internal_node, q: query center, r: search radius)
  {     parent_query_distance = d (N, q)
        for each child i of N
        {
                if ( |parent_query_distance - N.parent_child_distance(i) | > r + N.child_radius(i) )
                        prune child i

                child_query_distance = d (N.child(i), q)
                if (child_query_distance <= r + N.child_radius(i))
                        add child i to the list to be searched
                else
                        prune child i
        }
    }
```

The third algorithm for is for the large nodes which use all the three versions of the triangle inequality for pruning. The node contains its center, radius, references to its children, array of children's radii, array of parent-child distances, array of children's centers and also a 2-dimensional array containing distances between each pair of children. These distances are also pre-computed at the time of creation to save distance calculations from happening at search time. The space requirement for the large nodes is O (n^2) because of the 2-d array of children distances and the time requirement of the pruning algorithm is O (n^2) in the worst case.

Algorithm 3:

```
    visit_large_HP_internal (N: large_HP_internal_node, q: query center, r: search radius)
    {   parent_query_distance = d (N, q)
        list to_be_searched

        for each child i of N
        {       if ( |parent_query_distance - N.parent_child_distance(i) | > r + N.child_radius(i) )
                        prune child i
                else
                        add child i to the to_be_searched list
        }

        for each child i which is present in the list to_be_searched
        {       child1_query_distance = d (N.child(i), q)
                if (child1_query_distance > r + N.child_radius(i))
                        prune child i

                for each child j which is not equal to i and present in the to_be_searched list
                {       if ( |child1_query_distance - N.child_child_distance(i,j) | > r + N.child_radius(j) )
                                prune child j
                }
        }
        for each child i present in to_be_searched
                search child i
    }
```

Besides the above algorithms for pruning the internal nodes, the algorithm to search all the leaf nodes that have not been eliminated after the entire tree has been traversed, also uses a triangle inequality to attempt to prune the leaf node. If the leaf node is not discarded, it does a comparison of the distance from each child to the query against the radius and returns the appropriate data points as the search results. The leaf node stores its center, radius, references to the children, which are the data points, array of children's centers and an array of distances from the parent to each child. The space requirement for the leaf node is therefore O (n) and the complexity of the search is also O (n).

Algorithm 4:
visit_HP_leaf (N: HP_leaf_node, q: query center, r: search radius)

```
{      parent_query_distance = d (N, q)

       for each child i of N
       {          if ( | parent_query_distance - N.parent_child_distance(i) | > r )
                          prune the current node N

                  child_query_distance = d (N.child (i), q)
                  if (child_query_distance ≤ r)
                          return child i as a result
       }
}
```

VPT Proximity Search Algorithms

In VP internal nodes, a range is stored for the distances from the children to the vantage points, instead of an exact distance value being stored. Therefore each *VPInternalNode* contains an array of lower bounds of distances from the children to the first vantage point (*min_child_vp1*), an array of upper bounds of distances from the children to the first vantage point (*max_child_vp1*), an array of lower bounds of distances from the children to the second vantage point (*min_child_vp2*) and an array of upper bounds of distances from the children to the second vantage point (*max_child_vp2*). Using all this information, the pruning algorithm for the internal node of a vantage point tree consists of selecting those children whose bounds intersect with the proximity query after it has been mapped to the 2-dimensional space of distances from the two vantage points. The space requirement for the VP internal nodes is O (n) considering only 2 vantage points, but would be O (nm) if there were m vantage points. The complexity of the search is O (n) if the internal node has n children.

Algorithm 5:
visit_VP_internal (N: VP_internal_node, q: query center, r: search radius)

```
{      min_bound_query_vp1 = d (N.vantage_point1, q) – r
       max_bound_query_vp1 = d (N.vantage_point1, q) + r
       min_bound_query_vp2 = d (N.vantage_point2, q) – r
       max_bound_query_vp2 = d (N.vantage_point2, q) + r

       for each child i of N
       {          if ( min_bound_query_vp1 > N.max_child_vp1 (i) )
                          prune child i
                  else if ( max_bound_query_vp1 < N.min_child_vp1 (i) )
                          prune child i
                  else if ( min_bound_query_vp2 > N.max_child_vp2 (i) )
                          prune child i
                  else if ( max_bound_query_vp2 < N.min_child_vp2 (i) )
                          prune child i
                  else add child i in the list of nodes to be searched.
       }
}
```

For the leaf nodes of a VPT, the triangle inequality is used again to prune out data points that are not in the result set. The VPLeafNode consists of an array of distances from the children to the first vantage point (*child_vp1_distance*) and an array of distances from the children to the second vantage point (*child_vp2_distance*). These distances are computed at the time of the creation of the tree. The space requirement for a VP leaf node is therefore O (*n*) for the case of two vantage points. The complexity of the search is also O (*n*).

Algorithm 6:

```
visit_VP_leaf (N: VP_leaf_node, q: query center, r: search radius)
{    vp1_query_distance = d (N.vantage_point1, q)
     vp2_query_distance = d (N.vantage_point2, q)

     for each child i of N
     {        if ( | vp1_query_distance - N.vp1_child_distance(i) | > r )
                      prune child i
              else if ( | vp2_query_distance - N.vp2_child_distance(i) | > r )
                      prune child i

              child_query_distance = d (N.child (i), q)
              if (child_query_distance ≤ r)
                      return child i as a result
     }
}
```

5. Experimental Results and Analysis

The experimental results comprise of the generalized hyperplane tree search algorithms tested on trees constructed separately with each of the three types of internal nodes. The test dataset that is being used is the Yeast proteome sequence dataset. After bulkloading the trees using the hierarchical top-down bottom-up algorithm, the proximity search is performed with the yeast sequences as queries and specified radii. The query sequences are generated from the yeast dataset itself and are picked randomly based on a scale that is a parameter to the creation of the queries. Each of these proximity queries is then performed on the database with varying radii and results are returned. The parameters that can be varied in each search are the search radius, the database size and the number of queries. Statistics have been collected on the search and on the results. Result size is measured for each query as the rest of the statistics are correlated to the result size. The other indicators of search performance that are measured include the search time, number of distance calculations performed, the fraction of the total number of nodes that were visited, fraction of the total number of leaves that

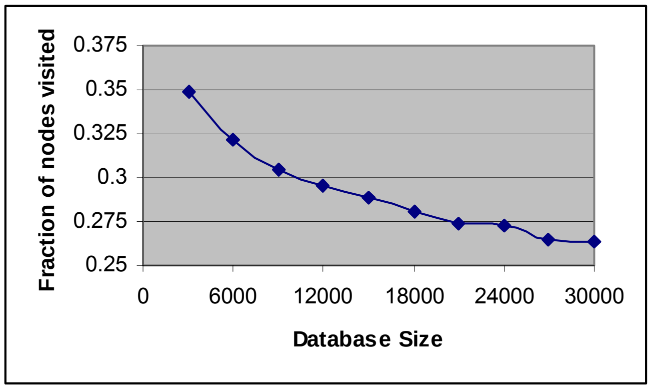


Fig 1: Fraction of nodes visited vs. Database size for medium nodes and radius 10

were visited, etc. On each of these statistics, the average, minimum, maximum values and the variance were recorded.

The first results in Fig 1 are those of the fraction of the total number of nodes visited as the database size grows from 3000 to 30000. The search is conducted with a radius of 10 and the node type is *MediumHPInternalNode*. Results indicate that for a database of size 3000, 35% of all the nodes are visited, and for a database of size 30000, about 26% of the medium nodes are accessed. This shows that the overall fraction of internal nodes visited in the tree is quite low and decreases as the data size grows.

In general, the result size varies from 0 to 5 for databases of size 3000 to 30000 for a search radius of 10. This exhibits the sparseness of the results in the data set that is being tested. This nature of the data hurts the search performance a lot as the data is very scattered. The next figure (Fig 2) exhibits the trend of the time it takes to search a database tree made up of medium HP internal nodes as the radius increases from 0 to 10. The search is done on a database of size 3000. The search time is in general a linear progression with the radius.

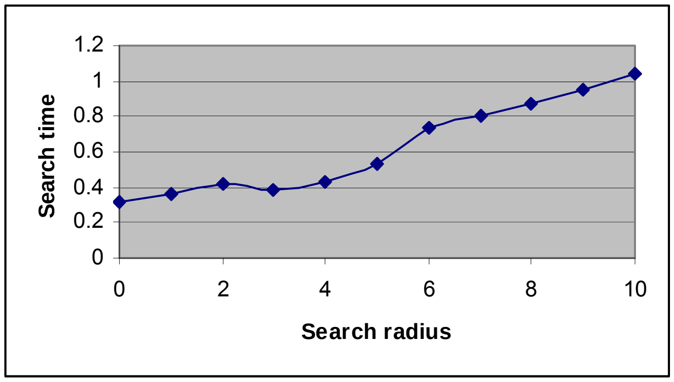


Fig 2: Time taken to perform a search query vs. search radius for a medium database of size 3000

The next set of experimental results includes a comparison of the various types of node structures and algorithms with respect to their search performance. Figures 3, 4 and 5 show the results of a database search of 100 queries of each of databases of sizes 3000, 6000 and so on till 30000. The search was conducted using all three types of algorithms: algorithm 1 which uses the first triangle inequality (small HP internal node), algorithm 2 which uses a combination of the first and second triangle inequalities

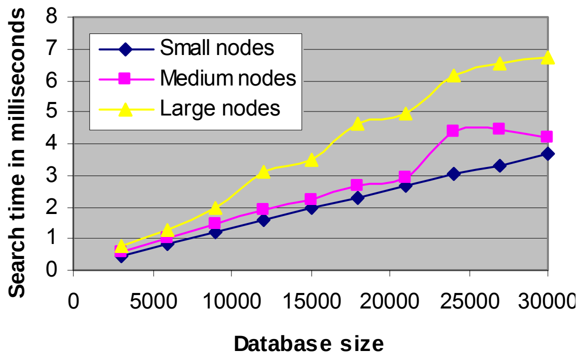


Fig 3: Search time vs. database size for all three algorithms

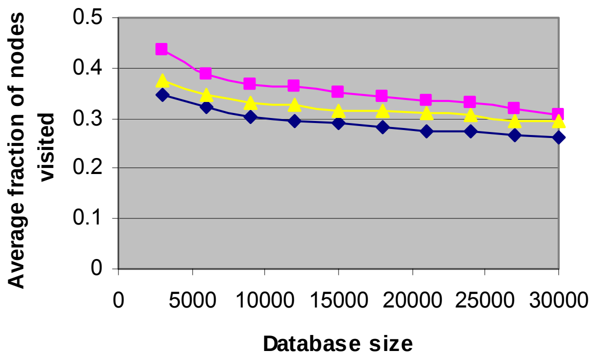


Fig 4: Average fraction of internal nodes visited vs. database size for all three algorithms

(medium HP internal nodes) and algorithm 3 which uses all the three triangle inequalities (large HP internal nodes). In the first graph, the search time has been plotted against the database size and it can be observed that the search time is minimum for algorithm 1 and maximum for algorithm 3. The average fraction of nodes visited from the second graph gives different performance results with respect to another parameter. Here the best performance is again given by the algorithm 1 as it has the minimum fractions. The highest fraction of nodes visited is for algorithm 3. The last parameter for analyzing the three different algorithms is the number of distance calculations, which is presented in Fig 5. Once again, the best performance is exhibited by algorithm 1. The results of the other 2 algorithms are fairly close, with the numbers for the large nodes being slightly better.

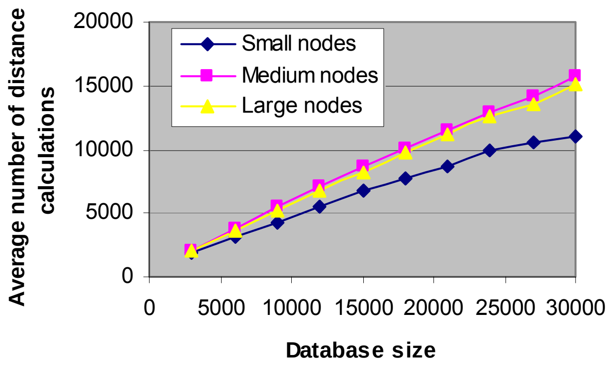


Fig 5: Average number of distance calculations vs. database size for all three algorithms

The results of this comparison of all the three types of algorithms for a generalized hyper plane search indicate that the best performance is exhibited by the first algorithm, which uses only triangle inequality 2 as its pruning criterion. The better performance of the small HP internal nodes can be attributed to the initial pruning criterion applied to the parent node. The initial condition *(parent_query_distance* $> r +$ *N.radius)* is unique in the first implementation and prunes out a parent node after entering it. The node structures using this algorithm also store the least number of data items and hence their space requirement is the least.

6. Conclusion and Future Work

The performance of the proximity search algorithms is dependent on a variety of factors such as the tree structure, the pruning criterion, the dimensionality of the data set, the quality of clustering of the data and the tree initialization algorithm. Among these, the factors that were being controlled and studied in this work were the tree node structures and the elimination rules. After developing various algorithms and measuring their performance, the experimental results demonstrate that the algorithm using the second triangle inequality, $| d(P, Q) - d(P,C) | > r(Q) + r(C)$, has the best performance both in terms of time and space. The performance results of proximity search in biological databases are constrained because of the dimensionality of the data, which is inherent in the nature of the data [3, 4]. Also retuning multiple data points as the results of a single query has a serious impact on performance because all possible paths in the tree that could potentially lead to an answer have to be traversed, unlike normal range search queries where the search can terminate after receiving a single valid answer. For this reason, the clustering of the data which happens in the tree creation algorithm is critical to the search. The better the clustering, the more localized the search can be made in order to produce better results.

Further research will progress in the direction of exploring more types of tree indexing algorithms and hence trying the search with different node structures. Also, other combinations and versions of the triangle inequalities can be tried. In the future, experiments will be conducted with the vantage point tree structure and different strategies will be tried with VPT. One possibility is a VPT containing multi-vantage points, instead of just two as is currently done. Also, other parameters of the tree construction and search can be varied to see their effects on the search. For example, the number of children in each node, which is determined by the choice of the page size and occupancy, is an important factor in the search. Other values could be tried for those parameters to see their effect. New search algorithms will be written as new methods to cluster the data and construct the tree are developed.

7. References

[1] T. Bozkaya and M. Ozsoyoglu. Distance-*based indexing for high-dimensional metric spaces*. In Proc. ACM SIGMOD International Conference on Management of Data, pages 357-368, 1997.

[2] S. Brin. *Near neighbor search in large metric spaces*. In Proc. 21st Conference on Very Large Databases (VLDB '95), pages 574-584, 1995.

[3] E. Chavez and G. Navarro. *Measuring the dimensionality of metric spaces*. Technical Report TR/DCC-00-1, Department of Computer Science, University of Chile, 2000.

[4] E. Chavez, J. Marroquin and G. Navarro. *Overcoming the curse of dimensionality*. In European Workshop on Content-Based Mutimedia Indexing (CBMI ’99), pages 57-64, 1999.

[5] E. Chavez and J. Marroquin. *Proximity queries in metric spaces*. In R. Baeza-Yates, editor, Proc. 4th South American Workshop on String Processing (WSP ’97), pages 21-36. Carleton University Press, 1997.

[6] E. Chavez, G. Navarro, R. Baeza-Yates, and J.L. Marroquin. *Searching in Metric Spaces* (Survey). ACM Computing Surveys, 2001.

[7] T. Chiueh. *Content-based image indexing*. In Proc. of the 20th Conference on Very Large Databases (VLDB ’94), pages 582-593, 1994.

[8] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns*. Addison-Wesley Publishing Company, 1995.

[9] Dan Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge University Press, 1997.

[10] Rui Mao, Weijia Xu, Neha Singh, Daniel P. Miranker. *An Assessment of a Metric Space Database Index to Support Sequence Homology*. 2002.

[11] J. K. Uhlmann. *Implementing metric trees to satisfy general proximity/similarity queries*. Manuscript, 1991.

[12] J. K. Uhlmann. *Satisfying General Proximity / Similarity Queries with Metric Trees*. Information Processing Letter, 40(4):175-179, 1991.

[13] P. Yianilos. *Data structures and algorithms for nearest neighbor search in general metric spaces.* In Proc. 4th ACM-SIAM Symposium on Discrete Algorithms (SODA ’93), pages 311-321, 1993.