AN EXAMINATION OF FAST SIMILARITY SEARCH TREES WITH GATING

A Thesis presented to
the Computer Science Department
at the University of Missouri

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

by

SETH J. WIESMAN
Dr. Jeffrey Uhlmann, Thesis Supervisor

MAY 2016

The undersigned, appointed by the Dean of the Graduate School, have examined the thesis entitled:

AN EXAMINATION OF FAST SIMILARITY SEARCH TREES WITH GATING

presented by Seth J. Wiesman,

a candidate for the degree of Master of Science and hereby certify that, in their opinion, it is worthy of acceptance.

Dr. Jeffrey Uhlmann

Dr. Rohit Chadha

Dr. Sean Goggins

ACKNOWLEDGMENTS

I would like to express my gratitude to my supervisor, Dr. Jeffrey Uhlmann, whose expertise, understanding, and patience, added considerably to my graduate experience. I have been amazingly fortunate to have an advisor who gave me the freedom to explore on my own, and at the same time the guidance to recover when my steps faltered.

Finally, I must express my very profound gratitude to my parents for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

TABLE OF CONTENTS

LIST OF FIGURES

List of Algorithms

ABSTRACT

The emergence of complex data objects that must be indexed and queried in databases has created a need for access methods that are both generic and efficient. Traditional search algorithms that only check specified fields and keys are no longer effective. Tree-structured indexing techniques based on metric spaces are widely used to solve this problem. Unfortunately, these data structures can be slow as the computational complexity of computing the distance between two points in a metric space can be high.

This thesis will explore data structures for the evaluation of range queries in general metric spaces. The performance limitations of metric spaces will be analyzed and opportunities for improvement will be discussed. It will culminate with the introduction of the Fast Similarity Search Tree as a viable alternative to existing methodologies.

Chapter 1

Introduction

1.1 Motivation

In the age of big data, knowledge discovery is no longer bottle-necked by the amount of data that can be collected but instead upon its quality and the speed at which it can be processed and queried. In domains as diverse as multimedia, molecular biology, and graphics, the number of data resources available is growing rapidly, both in terms of database size and the variety of forms in which data comes. While searching has always been one of the most prominent data processing operations, exact-match retrieval, typical for traditional databases, is neither feasible nor meaningful for modern data types. To cope with the onslaught of information, it is vital to have tools to search these resources efficiently and effectively.

Traditional retrieval techniques, based upon sorting and hashing, are not appropriate for a growing number of domains. Consider a collection of colors. We can sort colors according to their similarity with respect to a certain hue, such as red. But we cannot sort the set of all colors in a way such that for each hue its immediate neighbor is the hue most similar to it.

Instead of exact matches, a useful search paradigm is to quantify the similarity of a query point versus the points stored in a database. Points that are near a given query point form the query result set. Formally, the search problem is modeled in metric space. The collection of objects to be searched forms a subset of the metric space's domain and the distance applied to pairs of objects is a metric distance.

In a metric space, the only operation on data points is the evaluation of a distance function of a pair of points which satisfies the triangle inequality. Contrast this to points in a coordinate space – a special case of metric spaces – where points can be viewed as vectors. Such spaces satisfy some addition properties that can be exploited in indexing structures. Along with calculating the distance between two vectors other operations include vector addition and subtraction, allowing new vectors to be constructed from old ones[1, 2]. The k-d tree and range tree are two well known coordinate space data structures.

Because many data sets can be represented by vectors, it may appear unnecessary to to focus on pure metric spaces where the number of operations is limited. However, applications managing non-vector data like images and text do exist, and their number is growing. But even when the points are vectors, the properties of the underlying coordinate space cannot always be easily exploited.

If the individual vectors are correlated, the points seen through the lens of the distance measure between them will not map directly to their coordinate space and back. For instance, when searching images using color similarity, cross talk between colors exist and must be taken into account.

Another issue with traditional solutions for searching is that they suffer from the *curse of dimensionality*[3]. Such structures can become as slow as naive brute force search algorithms as the dimensionality increases. And while indexing structures do currently exist for organizing data in metric spaces, they are typically slow as they require many calls to the distance function which may be very computationally expensive.

This thesis will provide an examination of existing methodologies and then look at the Fast Similarity Search Tree as a superior alternative for performing range queries in general metric spaces.

1.2 Outline

The organization of this thesis is outlined as follows. Chapter 1 begins be providing motivation for the research proposed in this thesis. Work leading up to the research presented here is detailed in Chapter 2. This includes a look at the k-d tree as well as range tree, an introduction to metric spaces and a variety of metric functions, followed by an examination of the metric tree. Chapter 3 will detail the mathematical underpinnings of the Fast Similarity Search Tree as well as its implementation. Chapters 4 will look at how the Fast Similarity Search Tree fairs in practice on a variety data sets and Chapter 5 will provide a final discussion and set a course for future work.

Chapter 2

Background Information

2.1 Range Search

A range search is a geometric query where all the points in a region are reported[4]. Searching for a range leads to more than one value being found. The most basic type of range query is the orthogonal range query. Orthogonal range searching is a direct generalization of list searching. Let $P = \{p_1, \dots p_n\}$ be a set of n points in d dimensional space. Given a query $q = [min_1, max_1] \times \cdots \times [min_d, max_d]$, report all the points in $P \cap q$[5].

An example of such a search would be given a collection of students at a school find all those aged 18 - 21 and who have gpa's in the range 3.2 - 3.75.

2.1.1 One Dimension

Before generalizing to multiple dimensions, we will look at a range search on 1-dimensional data. In a single dimension the data set can be viewed as points on a line, and the query as an interval. The simplest 1-dimensional data structure for range searching is the unsorted list which permits a brute force query time of $\mathcal{O}(n)$[6]. If the data is sorted then the first element of the range can be found by using binary search and then the rest of the range by iterating until the first element outside of the range is found; reducing the complexity to $\mathcal{O}(log(n) + k)$[7]. Unfortunately, this method does not generalize to higher dimensions nor does it allow efficient updates to the data set.

Binary Search Tree

Instead of a list we can use a more sophisticated data structure, a balanced binary search tree[7]. Range searches on a binary search tree are performed by iterating down the tree until hitting a node where the path to the two endpoints of the query interval split. After this node is found, the search continues down both the left and right subtrees, comparing nodes to the endpoints. When comparing to the lower end point the search path goes to the left and all objects in that nodes subtree are reported. Searches down the right subtree follow a similar pattern but searching goes down the right and all objects to the left are reported. Figure 2.1 shows an example of searching for all values in the range $[17, 50]$.

The complexities for a balanced binary search tree are well known; it uses $\mathcal{O}(n)$ storage and can be built in $\mathcal{O}(n * log(n))$ time[6]. Because the search key is a range the result set is also a range and so the search time is based not only on the time taken

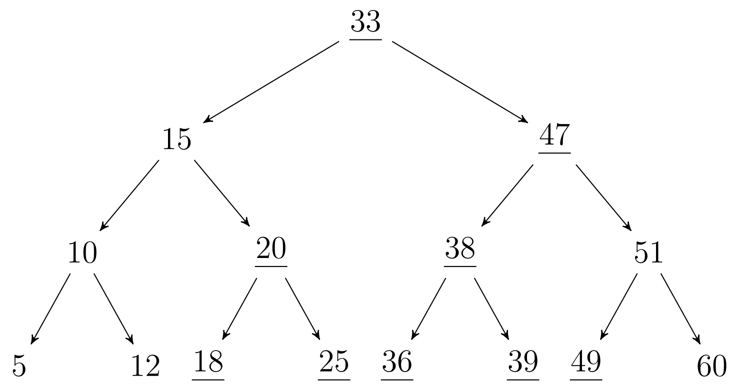

Figure 2.1: A range search on a binary search tree.

to find the ends of the interval but also the time to collect all the keys in between. If it takes $\mathcal{O}\left(log(n)\right)$ time to find the edges of the interval and k keys are reported back then the search will take $\mathcal{O}\left(log(n) + k\right)$ time[8]. Assuming $k << n$, this is near logarithmic.

2.1.2 Multiple Dimensions

Having solved range queries in one dimension, we can now generalize to higher dimensions.

2.1.3 K-D Tree

Extending a binary search tree to multiple dimensions results in the k-d tree[9]. This is a space-partitioning data structure for organizing points in d-dimensional space. Suppose a k-d tree were built with points in 2-dimensional space, then it would be able to quickly retrieve all points where $x \in [x_{min}, x_{max}]$ and $y \in [y_{min}, y_{max}]$.

The tree is built by first splitting the points into two subsets of roughly equal size on the x-axis. The point on the splitting line becomes the root, those points to the left of the line go into the left subtree and the rest go into the right. At the left child of the root the points are split into two subsets with a horizontal line on the y-axis; the points below or on it are stored in the left subtree of the left child, and the points above it are stored in the right subtree of the left subtree. Those points in the left child itself stores the splitting line. Similarly, the right subtree is split with a horizontal line into two subsets, which are stored in the left and right subtrees of the right child. At the third level the points are repartitioned on the x axis and the process is repeated until there are no points left to split. In the general case, at each level the set of points are partitioned by a hyperplane perpendicular to the x_i axis.

Searching a k-d tree is analogous to searching a binary search tree. Let P be a set of n points in 2-dimensional space. If we are searching for all the points in the range $[x_{min}, x_{max}] \times [y_{min}, y_{max}]$ then a point p lies within the query if

$$p_x \in [x_{min}, x_{max}] \wedge p_y \in [y_{min}, y_{max}]$$

We could say this single 2-dimensional range query is composed of 2 1-dimensional range queries. In general, an orthogonal range query in d-dimensions can be decomposed into d 1-dimensional range queries.

The search consists of a series of comparisons with median values, but now we alternate between the different dimensions. In the two dimensional case, if x_{min} is less than the median value x coordinate, then the left subtree must be examined. If x_{max} is greater than the median value of x, then the right subtree must be examined. At the next level, the comparison is performed using y_{min} and y_{max} respectively. Only points in leaf nodes that are reached mush be tested for membership in the search range.

Because a d-dimensional k-d tree for a set of n points is a binary tree with n leaves, it uses $\mathcal{O}\left(n\right)$ storage and its construction time is $\mathcal{O}\left(n * log(n)\right)$. It can be shown that the query time is bounded by $\mathcal{O}\left(n^{1-\frac{1}{d}} + k\right)$. As the dimensionality increases the query time quickly approaches linear.

2.1.4 Range Tree

Because the k-d tree may not have satisfactory query time, it is necessary to look at other data structures. One such option is the range tree, which has better query time, namely $\mathcal{O}\left(log^{d-1}n + k\right)$[10]. The price we have to pay for this improvement is an increase in storage from $\mathcal{O}\left(n\right)$ to $\mathcal{O}\left(n * log(n)\right)$.

As observed before, a single d-dimensional range query is equivalent to d 1-dimensional range queries. The intuition behind the k-d tree was to split the point set on alternating attributes. To obtain the range tree, we will use this observation in a different way.

Like before, lets begin with the 2-dimensional case where we want to be able to satisfy a query in the range $[x_{min}, x_{max}] \times [y_{min}, y_{max}]$. We first focus on finding the points whose x-coordinate lies in the range $[x_{min}, x_{max}]$, and worry about the y-coordinate later. If we only care about 1 coordinate then the query becomes a 1-dimensional range query which can be answered efficiently using a balanced binary search tree.

Let's call the subset of points stored in the leaves of a subtree rooted at a node n the subset of n. The subset of the root of a tree is the whole set of points while the subset of a leaf is simply the point stored in that leaf. In the 1-dimensional case we are searching for the node n whose subset contains all nodes in our search range. As seen at the beginning of this chapter, that result can be found in $\mathcal{O}\left(log(n)\right)$ time.

Now, in the 2-dimensional case, we are not interested in the entire subset but only those points whose y-coordinate lies in the interval $[y_{min}, y_{max}]$. This is yet another 1-dimensional range query, which we can solve, provided we have a balanced binary search tree on the y-coordinate of the points of the subset of n. This leads to the following data structure for rectangular range queries on a set of points known as a range tree.

1. The main tree is a balanced binary search tree build on the x-coordinate of the points.

2. For any node n, its subset is stored in a balanced binary search tree on the y-coordinate of the points. The node n stores a pointer to the root of that tree.

Generalizing a range tree to higher dimensions is fairly straight forward. Given a set of points in d-dimensional space, we first construct a balanced binary search tree on the first coordinate of those points. The subset of a node n in the first level of the tree, consists of the points stored in the leaves of the subtree rooted at n. For each node n we construct an associated data structure; another range tree built on the subset of n restricted to their last d-1 coordinates. This (d-1)-dimensional range tree is recursively constructed the same way.

Searching a d-dimensional range tree requires performing d 1-dimensional range searches. First a range search is performed on the first dimension and a subtree rooted at n is found whose values all fall in the range on the first dimension. The process is then recursively repeated on each dimension until the final result set is found.

2.1.5 Limitations of Orthogonal Range Queries

K-d tree's and range tree's both provide methods for answering orthogonal range queries. Unfortunately, not all range queries are orthogonal. For instance, in a geographic database we may wish to ask the question "*What museums are within 2 miles of my hotel?*" More formally, we wish to solve the query $R(t, r)$ on a set of points S where the query is specified by a target t with some radius r as the distance constraint.

$$R(t, r) \leftarrow \{o \in S \mid d(t, o) \leq r\}$$

The first issue is that unlike an orthogonal range search where the query is rectangular, the shape of this query is circular. A circle can be approximated by a bounding box but as dimensionality increases the volume between the circle and the box will grow[11]. This can lead to a large number of false positives or us having to then perform a second brute force range search on the initial result set.

Another issue is that the k-d tree and range tree only work on coordinate spaces, where the value at each dimension can be independently examined. In many contexts, such as image, text, or dna similarity, the concept of dimensionality may not make sense. This leads us to introducing the idea of metric spaces and a more general search structure.

2.2 Metric Space

Suppose the pair $\mathcal{M} = (D, d)$ where D is a set and d is a real function defined by the Cartesian product $D \times D \rightarrow \Re$ [12]. Then d is called a metric on D if, and only if:

$\forall x, y \in D, d(x, y) \geq 0$	non-negativity
$\forall x, y \in D, d(x, y) = d(y, x)$	symmetry
$\forall x, y \in D, x = y \Leftrightarrow d(x, y) = 0$	identity
$\forall x, y, z \in D, d(x, z) \leq d(x, y) + d(y, z)$	triangle inequality

Figure 2.2: Metric Space Properties

The expression d(x,y) is read “the distance from x to y.” When it is unambiguous we simply say that D is a metric space, and the distance function d is a metric.

The value of metric spaces is that it is unnecessary to find a meaning for the data with respect to the axes of a coordinate system. The only way for sets of points to be compared is to make use of the properties listed above[13]. Metric space indexes are usually built off-line while queries are executed on-line.

2.2.1 Metric Functions

The distance function of a metric space represents a way of quantifying the closeness of objects in a given domain. In the following section we will examine several examples of distance functions used in practice on various types of data. Distance functions are often tailored to specific applications or a class of possible applications. In practice, distance functions are specified by domain experts.

Minkowski Distances

The Minkowski distance metrics form a family of distance functions[14]. They are called the L_p metrics because the individual cases depend on the numeric parameter

p and are defined on d-dimensional vectors of real numbers as:

$$L_p\left[(x_1,\dots x_d)(y_1,\dots y_d)\right] = \sqrt[p]{\sum_{i=1}^{d} \mid x_i - y_i|^p}$$

Some of the metrics are:

1. L_1 - Manhattan Distance: The distance metric used in the popular A^* path finding algorithm [15]

2. L_2 - Euclidean Distance: Undoubtedly the most widely known distance measure is euclidean distance. This is the measure of the straight line distance between two points.

3. L_∞ - Chebyshev Distance: A metric defined on a vector space where the distance between two vectors is the greatest of their differences along any coordinate dimension: $max\left\{x_i - y_i\right\}$. It is commonly called the chessboard distance because in the game of chess the Chebyshev distance is equal to the number of moves required to move the king from one square to another[16].

Quadratic Form Distance

Many applications using feature vectors have individual components, features, that are correlated. Consider color histograms of images, where each dimension represents a specific color. When computing a distance, the red component must be compared not only with the dimension representing red, but also with pink and orange because those colors are similar. If a euclidean distance, L_2, were used the resulting distance would not reflect any correlation amongst the features. The quadratic form distance is a metric which accounts for these correlations[17].

The distance between two d-dimensional vectors, x and y, is based on the strength of the correlation between features. This correlation is provided by an $n \times n$ covariance matrix C where the weight of C_{ij} denotes the strength of the correlations between x_i and y_j. The weights are usually normalized so that $0 \leq C_{ij} \leq 1$ with diagonal elements equal to 1.

$$d(x, y) \leftarrow \sqrt{(x - y)^T * C * (x - y)}$$

Note that when M is equal to the identity matrix, this distance becomes euclidean distance.

Levenshtein Distance

Also known as edit distance, Levenshtein distance can be used to measure the difference between two strings which may be useful in applications such as spell checking. The distance is defined as the minimum number of edit operations – insertion, deletion, and substitutions – necessary to convert one string into another[18].

Tree Edit Distance

Tree edit distance defines the distance between two trees as the minimum cost to convert one tree into another through insertion and deletion of nodes. The cost of edit operations may be constant for the whole tree, or may vary with the level in the tree were the operation is carried out. This is because the adding a node near the root may be more significant than adding one near a leaf. XML documents are typically modeled as rooted trees so tree edit distance can be used to measure their structural distance[19, 20].

Jaccard Distance

The Jaccard similarity is a measure of the closeness of sets. It is not really a distance as the closer the sets are the higher the similarity.

$$J(A, B) = \frac{A \cap B}{A \cup B} = \frac{\mid A \cap B \mid}{\mid A \mid + \mid B \mid - \mid A \cap B \mid}$$

Instead, 1 minus the Jaccard similarity is a distance measure, called the Jaccard distance[14].

$$J_\delta(A, B) = 1 - J(A, B)$$

As an example application, suppose a retailer has a log file containing the purchases of its customers. Jaccard's distance can be used to find similar customers and a shopper could be offered coupons based on what other shopper's like them have bought in the past.

2.2.2 Metric Function Time Complexities

In general, computing the distance between two items is a nontrivial process. For example, the L_p norms are computed in linear time dependent on the dimensionality d of the vector space. The quadratic form distance is much more expensive because it involves multiplication by a covariance matrix C. Returning to the color image example, color image histograms are typically high-dimensional vectors consisting of 64 or 256 distinct colors or vector dimensions.

Edit distance between two strings of with lengths n and m have complexity

$\mathcal{O}(nm)$[21]. Tree edit distance is even more complex and has a worst case complexity of $\mathcal{O}(n^4)$, where n is equal to the number of tree nodes[22].

In brief, due to the high computational complexity of metric distance functions, it is important for metric index structures to minimize the number of distance calculations they perform.

2.2.3 Ball Decomposition

Because objects in metric spaces are not guaranteed to have discrete features to compare against, a new partition strategy is necessary. One such option is the ball decomposition. A ball decomposition is a partitioning strategy where a set of objects are partitioned according to whether they are inside or outside of a specified ball-type region[23]. Ball decomposition breaks a set of objects S in a metric space $\mathcal{M} = (D, d)$ into two sets S_1 and S_2 using a random pivot p.

$$d_{median} \leftarrow \underset{o \in S}{median}\,\{d(o,p)\}$$

$$S_1 \leftarrow \{o \mid o \in S \setminus \{p\} \wedge d(o,p) \leq d_{median}\}$$

$$S_2 \leftarrow \{o \mid o \in S \setminus \{p\} \wedge d(o,p) \geq d_{median}\}$$

Partitioning based on the median distance ensures the two sets are of equal size.

2.2.4 Metric Tree

The metric tree is a data structure for organizing objects in a metric space[23]. It is based on the ball decomposition method described in the previous section, which

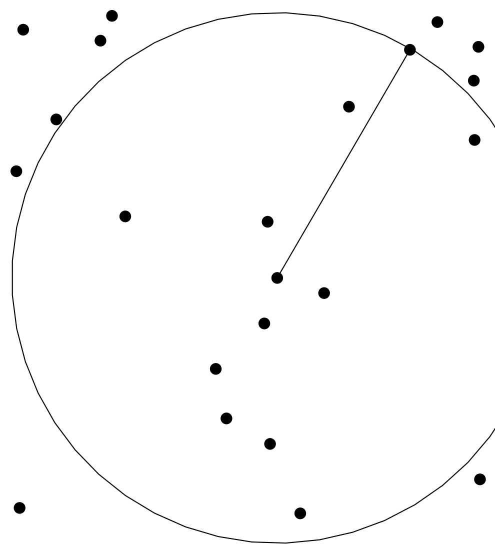

Figure 2.3: Ball Decomposition

divides a set S into subsets S_1 and S_2 based upon a pivot object p and the median distance d_{median} from p to all other objects in S. Beginning with the entire set of objects S and recursively applying ball decompositions results in a balanced binary tree. Each node contains its pivot point and associated d_{median}. The objects in S_1 go into the left subtree while S_2 goes to the right subtree. See figure 2.3.

The search algorithm for a range query $R(t, r)$ traverses the metric tree from root to leaves. For each node, it evaluates the distance $d(t, p)$ between the pivot and the target. If $d(t, p) \leq r$ then the pivot p is reported. For internal nodes, the algorithm must also decide which subtrees to search. This requires setting a lower bound on the distance from t to objects in the left and right subtrees. If the query radius r is less than the lower bound, then the corresponding subtree does not have to be visited. The

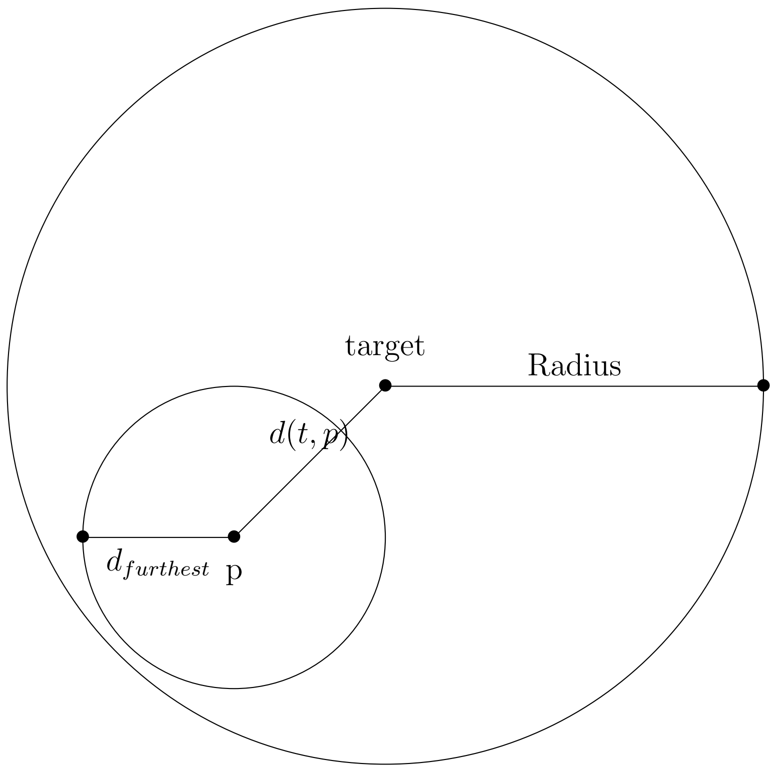

Figure 2.4: Fully Enclosed Subtree

distance from t to any object in the left subtree is at least $d(t,p) - d_{median}$. Likewise, the distance from t to any object in the right subtree is at least $d_{median} - d(t,p)$. Thus, the left subtree is visited if $d(t,p) - d_{median} \leq r$ and the right subtree is visited if $d_{median} - d(t,p) \leq r$. Notice that if the radius is large enough, both subtree's may be visited.

If it can be shown that the query space fully contains a subtree of the current node, such as figure 2.4, then all the points in that subtree may be collected. Let $d_{furthest}$ be distance from the current point to the furthest point from it in a particular subtree. Then that subtree is fully contained by the query if $d(t,p) + d_{furthest} \leq r$[24].

2.2.5 Metric Tree Limitations

While metric tree's do solve the problem of range queries in general metric spaces they have certain drawbacks. During construction many distance calculations between pivots and objects are computed. For every object in the tree distances are computed from it to each pivot p on the path from root to its node, however only the last of these results are maintained after the tree is built.

Metric functions are typically costly to compute and the metric tree may make a large number of distance calculations while performing a search. The search algorithm does not take any advantage of previously calculated distance measures, this means that no free checks are given.

Chapter 3

Fast Similarity Search Tree

The Fast Similarity Search Tree, FaSST, is a balanced binary tree data structure optimized for efficient range query's over general metric spaces[25]. Like the metric tree, FaSST works with any metric space. In the same way the range tree improved on the k-d-tree with the use of extra memory, this structure improves on the metric tree at the cost of $\mathcal{O}\left(()\, n * log(n)\right)$ space. It is built similarly to a metric tree – by recursively performing ball decompositions – but now each point retains the distance from itself to each of the pivots above it. By finding the distances of the closest and furthest neighbors in each subtree to each of the pivots above them, a more aggressive pruning strategy may be used. Additionally, these distances can be used to implement a gating mechanism which can allow points to be included or excluded from the result set without having to perform distance calculations.

3.1 Methods for avoiding Distance Calculations

Because distance calculations can be expensive, computing any more than necessary should be avoided. Due to properties of functions in metric spaces, primarily the triangle inequality, there are several ways to avoid these computations.

3.1.1 Bounded Search Spaces

The first constraint independently bounds the search space of the subtrees of each node. The goal is to form a volume containing all of the nodes in a particular subtree. If the query $R(t, r)$ is viewed as a hypersphere centered at t with radius r, then it can be thought that a subtree should only be visited if the hypersphere intersects its volume.

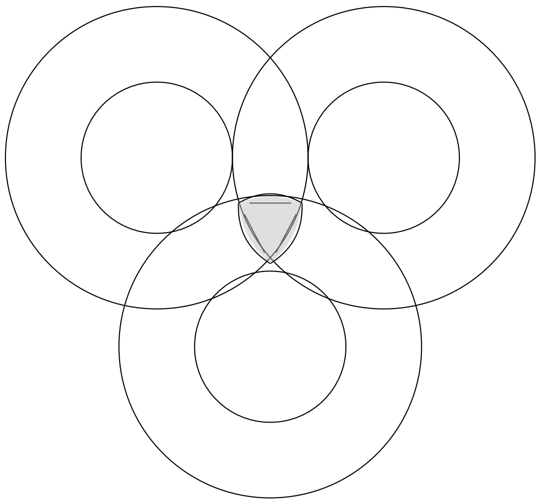
Figure 3.1: Annulus Intersection

Given a node at depth d, d reference points $\{p_1, p_2 \ldots p_d\}$ are chosen. These reference points are the pivot points above the current node plus the point at the current node. For each subtree of a node, each reference point p_i is examined. The nodes of the subtree are bound inside h annular regions $\{A_1, A_2 \ldots A_d\}$ centered at each of the reference points. The short and long radii of a particular annulus are the distances from the reference point to its nearest and furthest neighbors in the subtree respectively. The final search space of a particular subtree consists of the intersections of all the annular regions.

Lemma 3.1.1. *Given a subtree bounded by a set of annular regions $\mathcal{A} = \{A_1, A_2 \ldots A_d\}$ and a query $\mathcal{R}(t, r)$ then the subtree only needs to be searched if:*

$$\mathcal{R}(t, r) \cap \left(\bigcap_{i=1}^{d} \mathcal{A}_i \right)$$

Similar to the metric tree, it may be possible to determine while searching that all points in a subtree are contained within the query space. Again, in this case, all points may be returned without further examination. Because all the points in a subtree are fully contained in every annulus, it is sufficient to find a single annulus fully contained in the search space. An annulus is fully contained if the distance from its center plus the long radius is less than the radius of the query.

Lemma 3.1.2. *Given a subtree bounded by a set of annular regions $\mathcal{A} = \{A_1, A_2 \ldots A_d\}$ and a query $\mathcal{R}(t, r)$ then all points in the subtree may be returned if:*

$$\exists A \in \mathcal{A} \mid d(A_{center}, t) + A_{longRadius} \leq r$$

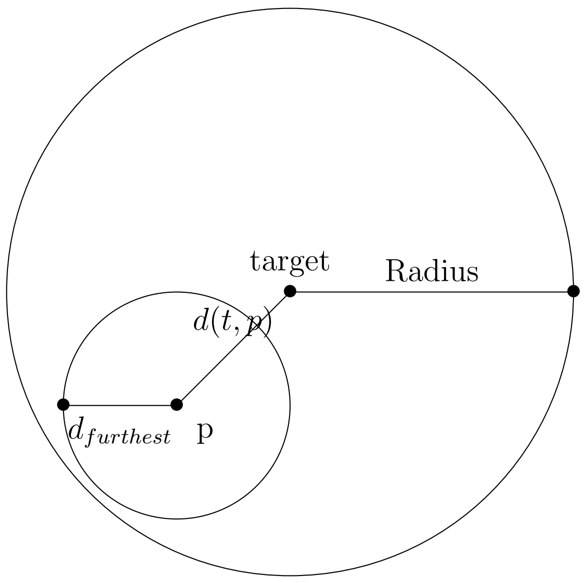

Figure 3.2: Fully Enclosed Subtree

3.1.2 Gating

The second constraint uses precomputed distances between the point at the current node and the pivots above it to prune points without performing any new distance calculations. Since these distances are computed at build time, this is computationally cheap to perform.

Lemma 3.1.3. *Given a metric space $\mathcal{M} = (D, d)$, then $\forall t, p, o \in D$, it is guaranteed:*

$$\mid d(t,p) - d(p,o) \mid \leq d(t,o) < d(t,p) + d(p,o)$$

Noted above, the distance between a pivot point p and current point o is computed and stored at build time. As well, the distance between the query target t and the pivot point p will have already been calculated when p was originally visited. Because

$| d(t,p) - d(p,o) | \leq d(t,o)$, we know that if the range of the query $r < | d(t,p) - d(p,o) |$ then the point can be automatically excluded from the result set. Therefore, $d(t,o)$ does not need to be calculated and one distance calculation is saved.

On the other hand, if $d(t,p) + d(p,o) \leq r$ then it is certain that $d(t,o) \leq r$ and o can be automatically included in the result set. Only if

$$r \in [| d(t,p) - d(p,o) |, d(t,p) + d(p,o))$$

does a distance calculation have to be made.

When building the data structure each node is compared not only to the pivot directly above it, but all the pivot nodes going back to the root of the tree. By retaining all of these measurements, each node can be compared to approximately $log(n)$ pivot points.

As the query point traverses down the tree it will collect its distance to each of the pivot points it passes. When a point is being tested for inclusion in the result set, there are now $\mathcal{O}(log(n))$ triangles that can be used to compute bounds. This operation is powerful because the triangle used to find the lower bound does not have to be the same one used to find the upper bound. This means the bounds can be optimized separately to form the smallest range possible.

Lemma 3.1.4. *Given a metric space $\mathcal{M} = (D, d)$, then $\forall t, o \in D$, and a collection of pivot points it is guaranteed:*

$$\underset{p_1 \in pivots}{max} \{| d(t,p_1) - d(p_1,o) |\} \leq d(t,o) < \underset{p_2 \in pivots}{min} \{d(t,p_2) + d(p_2,o)\}$$

Unfortunately, gating does not come without its issues. If a point is gated then the distance from that point to the query object is never calculated which may cause two issues. The first is that none of the children of that point will be able to use the triangle containing that point when gating. The second is that this is also the distance from t to the annulus centered at the gated node. As such that annulus may no longer be used when pruning any of the subtree's further down the tree. This may lead to the search not terminating as early as it may otherwise.

3.2 Implementation

The rest of this chapter contains implementation details for the Fast Similarity Search Tree. For the remainder of this thesis, FaSST will refer to a binary tree that implements pruning using the bounded search space method. FaSST with Gating will refer to a tree that implements pruning using the bounded search space method and gating on each node.

3.2.1 Construction

The tree is constructed by first partitioning the points using a ball decomposition. First, a random point is chosen to be the pivot and becomes the root of the tree. The distance from the pivot point to all other points is calculated and the distances are saved for latter use. The median distance from the root is found and the nodes are partitioned such that all those less than the median are in the left subtree and all those greater than the median are in the right subtree.

After partitioning the nodes, the search space of the left and right subtrees are independently bounded. The goal is to form a volume around all of the point in a subtree so that it only need be searched if the query space and the volume intersect. This is accomplished by using $\mathcal{O}\left(log(n)\right)$ reference points, specifically all of the points from the current node up to the root of the tree. Each of the reference points forms the center of an annulus where the short radius is the distance from the reference point to its nearest neighbor in the subtree and long radius and the distance from the reference point to its furthest neighbor in the subtree. The process is then recursively repeated until there are no points left to partition.

root:	The root of a FaSST
target:	The target point of the query
radius:	The radius of query
pivots:	The distance from the target to each of the previous pivot points
depth:	The current depth in the tree
d:	A metric distance function
annuli:	All the annuli surrounding a subtree

Figure 3.3: Pseudocode Variables

```
Output: The root of the tree
Build(nodes, depth, distance) begin
    if nodes.$empty()$ then
        return null
    end
    if nodes.$size() == 1$ then
        return $the remaining node in nodes$
    end
    root ←a random node ∈ nodes
    for node ∈nodes do
        node.pivots[depth ]←distance(node.$point$, root.$point$)
    end
    pivot ← $\text{median}_{node \in nodes \setminus \{root\}}\{node.pivots[depth]\}$
    left ← ∀ node ∈nodes \ {$root$} | node.ancestors[depth] < pivot
    right ←nodes \(left ∪ {$root$})
    root.left←Build(left, depth $+1$, distance)
    root.right←Build(right, depth $+1$, distance)
    for $i \in [0, \text{depth})$ do
        if root.$left$ ≠ null then
            nearest ← $\min_{node \in root.left}\{node.pivots[i]\}$
            furthest ← $\max_{node \in root.left}\{node.pivots[i]\}$
            root.leftAnnuli[i]←Annulus(nearest, furthest)
        end
        if root.$right$ ≠ null then
            nearest ← $\min_{node \in root.right}\{node.pivots[i]\}$
            furthest ← $\max_{node \in root.right}\{node.pivots[i]\}$
            root.rightAnnuli[i]←Annulus(nearest, furthest)
        end
    end
    return root
end
```

Algorithm 1: Build a Fast Similarity Search Tree

3.2.2 Search: FassT

When searching a FaSST, at any particular node we have access to:

1. The query target t and radius r.
2. The distances from the target to each of the pivots above; pivots[i] is equivalent to $d(q, p_i)$.
3. The candidate point at the current node that may or may not be included into the result set.
4. All the annuli for each of the current nodes subtrees.

A search begins by determining if the candidate point should be included into the result set by finding the distance from it to the target point. If at an internal node it must next be determined if either of the subtree's are fully enclosed by the query or if either intersect with it and should be searched.

The query fully encloses a subtree if the distance from the target point to the center of the annulus plus the long radius of that annulus is less than the query radius. Only a single annulus must be fully enclosed for the entire subtree to be returned.

If the query does not enclose the subtree then we must determine if it intersects with it. There are three conditions which indicate a query point intersects with an annulus. First, if the distance from a target point to the center of the annulus is between the short and long radii, then it automatically intersects. Secondly, if the distance from the target point to the center is less than the short radius then the query intersects if the distance from the center plus the search radius is greater than the short radius. Finally, if the distance from the target point to the center places it past the long radius, it intersects if that distance minus the search radius is less than the long radius of the annulus. If a single annulus does not intersect with the query than the entire subtree may be pruned.

This process is recursively repeated until search terminates.

```
Output: A set of all the objects in the search range
Search(root, target, radius, pivots, depth, d) begin
    if root == null then
        return $\emptyset$
    end
    result $\leftarrow \emptyset$
    distance $\leftarrow$d(target, root.point)
    if distance $\leq$ radius then
        result $\leftarrow$ result $\cup$ {root.point}
    end
    pivots [depth]$\leftarrow$ dist
    if Collect(root.leftAnnuli, pivots, radius) then
        result $\leftarrow$result $\cup$root.left
    end
    else if goLeft(root, target, pivots, depth) then
        result $\leftarrow$result $\cup$Search(root.left, target, radius, pivots, depth+1, d)
    end
    if Collect(root.rightAnnuli, pivots, radius) then
        result $\leftarrow$result $\cup$root.right
    end
    else if goRight(root, target, pivots, depth) then
        result $\leftarrow$result $\cup$Search(root.right, target, radius, pivots, depth+1, d)
    end
    return result
end
```

Algorithm 2: Search: FaSST

```
Output: True is the entire subtree may be collected, false otherwise
Collect(annuli, pivots, radius, depth) begin
    for $i \in [0, depth)$ do
        if $pivots[i] + annuli[i].longRadius \leq radius$ then
            return $true$
        end
    end
    return $false$
end
```
Algorithm 3: Collect Subtree: FaSST

```
Output: True if left subtree should be searched, false otherwise
goLeft(root, target, pivots, depth) begin
    if root.$left$ == null then
        return $false$
    end
    for $i \in [0, depth)$ do
        if pivots$[i] <$ root.$leftAnnuli[i].shortRadius$ then
            if pivots$[i] +$ radius $<$ root.$leftAnnuli[i].shortRadius$ then
                return $false$
            end
        end
        if root.$leftAnnuli[i].longRadius <$ pivots$[i]$ then
            if pivots$[i] -$ radius $>$ root.$leftAnnuli[i].longRadius$ then
                return $false$
            end
        end
    end
    return $true$
end
```
Algorithm 4: Prune Left Subtree: FaSST

```
Output: True if right subtree should be searched, false otherwise
goRight(root, target, pivots, depth, distance) begin
    if root.$right$ == null then
        return $false$
    end
    for $i \in [0, depth)$ do
        if pivots[$i$] < root.$rightAnnuli[i].shortRadius$ then
            if pivots[$i$] + radius < root.$rightAnnuli[i].shortRadius$ then
                return $false$
            end
        end
        if root.$rightAnnuli[i].longRadius$ < pivots[$i$] then
            if pivots[$i$] − radius > root.$rightAnnuli[i].longRadius$ then
                return $false$
            end
        end
    end
    return $true$
end
```

Algorithm 5: Prune Right Subtree: FaSST

3.2.3 Search: FassT with Gating

Searching a FaSST with Gating is identical to searching without gating except now the gating step must be performed before each distance calculation and the logic for deciding if the left or right subtree of a given node should be visited must be updated to handle the possibility of missing annuli.

The search function takes in the same arguments as before plus the distances from the current point to each of the pivots above; node.pivots[i] is equivalent to $d(node.object, p_i)$. Searching begins the same as before, by determining if the candidate point should be included into the result set. However this time, gating is used to potentially avoid a distance calculation. This is done by placing upper and lower bounds on its distance from the candidate point to the query point. A lower bound is set by finding the index i such that $max \{\mid nodes.pivots[i] - pivots[i] \mid\}$. If a particular $pivots[i]$ is unknown it shall be treated as $-\infty$. Finding the upper bound follows a similar process, an index i is found such that $min \{nodes.pivots[i] + pivots[i]\}$. If a particular $pivots[i]$ is unknown it shall be treated as ∞.

If the radius is less than the lower bound than the candidate point can be automatically included, and if it is greater than the upper bound it can be automatically excluded. If either of these conditions are met than the proper action should be taken and a sentinel value added to the query points list of pivot distances as to indicate that the true distance is unknown.

Otherwise, if the radius falls in the range then a distance calculation must be made and the point will be collect if the distance is less than or equal to the radius. Because a true distance was found it can be added to the query points list of pivots.

```
Output: A set of all the objects in the search range
Search(root, target, radius, pivots, depth, d) begin
    if root == null then
        return $\emptyset$
    end
    result $\leftarrow \emptyset$
    lowerBound $\leftarrow \max_{i \in [0,\text{depth})} \{|\text{ root}.pivots[i] - \text{pivots}[i]\ |\}$
    upperBound $\leftarrow \min_{i \in [0,\text{depth})} \{\text{root}.pivots[i] + \text{pivots}[i]\}$
    if radius $\notin$ [lowerBound, upperBound) then
        if upperBound $\leq$ radius then
            result $\leftarrow$ {root.point}
        end
        pivots [depth ]$\leftarrow \infty$
    else
        distance $\leftarrow$d(target, root.point)
        if distance $\leq$ radius then
            result $\leftarrow$ result $\cup$ {root.point}
        end
        pivots [depth]$\leftarrow$ dist
    end
    if Collect(root.leftAnnuli, pivots, radius) then
        result $\leftarrow$result $\cup$root.left
    end
    else if goLeft(root, target, pivots, depth) then
        result $\leftarrow$result $\cup$Search(root.left, target, radius, pivots, depth+1, d)
    end
    if Collect(root.rightAnnuli, pivots, radius) then
        result $\leftarrow$result $\cup$root.right
    end
    else if goRight(root, target, pivots, depth) then
        result $\leftarrow$result $\cup$Search(root.right, target, radius, pivots, depth+1, d)
    end
    return result
end
```

Algorithm 6: Search: FaSST with Gating

Determining whether a subtree should be fully collected or visited is again similar to before. The only difference is that if the distance from the target to a particular pivot is unknown, i.e. $pivots[i] == \infty$, then the current iteration of the loop must be skipped. In fact, the changes to the pruning functions is the if statement on lines 6-8.

```
   Output: True is the entire subtree may be collected, false otherwise
1  Collect(annuli, pivots, radius, depth) begin
2      for $i \in [0, depth)$ do
3          if $pivots[i] == \infty$ then
4              continue
5          end
6          if $pivots[i] + annuli[i].longRadius \leq radius$ then
7              return $true$
8          end
9      end
10     return $false$
11 end
```

Algorithm 7: Collect Subtree: FaSST with Gating

z

```
Output: True if left subtree should be searched, false otherwise
goLeft(root, target, pivots, depth) begin
    if root.left == null then
        return false
    end
    for $i \in [0, depth)$ do
        if pivots[$i$] == $\infty$ then
            continue
        end
        if pivots[$i$] < root.$leftAnnuli[i].shortRadius$ then
            if pivots[$i$] + radius < root.$leftAnnuli[i].shortRadius$ then
                return false
            end
        end
        if root.$leftAnnuli[i].longRadius$ < pivots[$i$] then
            if pivots[$i$] − radius > root.$leftAnnuli[i].longRadius$ then
                return fals
            end
        end
    end
    return true
end
```

Algorithm 8: Prune Left Subtree: FaSST with Gating

```
Output: True if right subtree should be searched, false otherwise
goRight(root, target, pivots, depth, distance) begin
    if root.$right$ == null then
        return $false$
    end
    for $i \in [0, depth)$ do
        if pivots[$i$] == $\infty$ then
            continue
        end
        if pivots[$i$] < root.$rightAnnuli[i].shortRadius$ then
            if pivots[$i$] + radius < root.$rightAnnuli[i].shortRadius$ then
                return $false$
            end
        end
        if root.$rightAnnuli[i].longRadius$ < pivots[$i$] then
            if pivots[$i$] − radius > root.$rightAnnuli[i].longRadius$ then
                return $fals$
            end
        end
    end
    return $true$
end
```
Algorithm 9: Prune Right Subtree: FaSST with Gating

Chapter 4

Results

In this chapter we experimentally evaluate the Fast Similarity Search Tree, both with and without gating. We compare these tree's to the k-d tree and metric tree in the case of vector spaces, and just metric tree in the case of general metric spaces. Code for all the data structures used is available on online on github at:

`https://github.com/sjwiesman/Metric-Space-Data-Structures` .

4.1 Correctness

To make sure that the answers retrieved from the queries are correct, every implementation has been tested against linear scan algorithms. As well as being throughly tested in advance, most tests done where the results are used in this thesis include a linear scan on every query to make sure that there are no errors. In the case of some of the more computationally heavy tests, this was not possible because of the extreme expense of doing a linear scan on the set of data objects or every query. Instead the results of each implementation were compared against one another and only used if every result set's were identical.

4.2 Vector Spaces

The first test compared the tree's in vector spaces. The tree's used are the k-d tree, metric tree, FaSST, and FaSST with Gating. Because the k-d tree cannot naively solve a radial range query, it found all points within a bounding box. A brute force search was then performed on all points within that box to find the final result set.

Tests were ran on uniformly distributed data sets in 2 - 10 dimensions. For each test in each dimensionality a radius was found that would return 5 points nearest to the origin using euclidean distance. The tree's started with 100 points and were rebuilt with the same points as before plus 100 new points up to 10000 points. The number of distance calls executed, number of nodes visited, and runtime in milliseconds are shown for each test.

4.2.1 Dimensions 1 - 4

In very low dimensionality the k-d tree performs well because the difference in volume between the a sphere and bounding box is minimal. The FaSST performs roughly equally and better than the metric tree. This is to be expected since it contains strictly more information than the metric tree and is able to prune more efficiently. The FaSST with Gating performs the worst in this space. This is because in low dimensions the gating performs to well and a lot of pruning information is lost which results quite a few more nodes having to be visited than the other tree's.

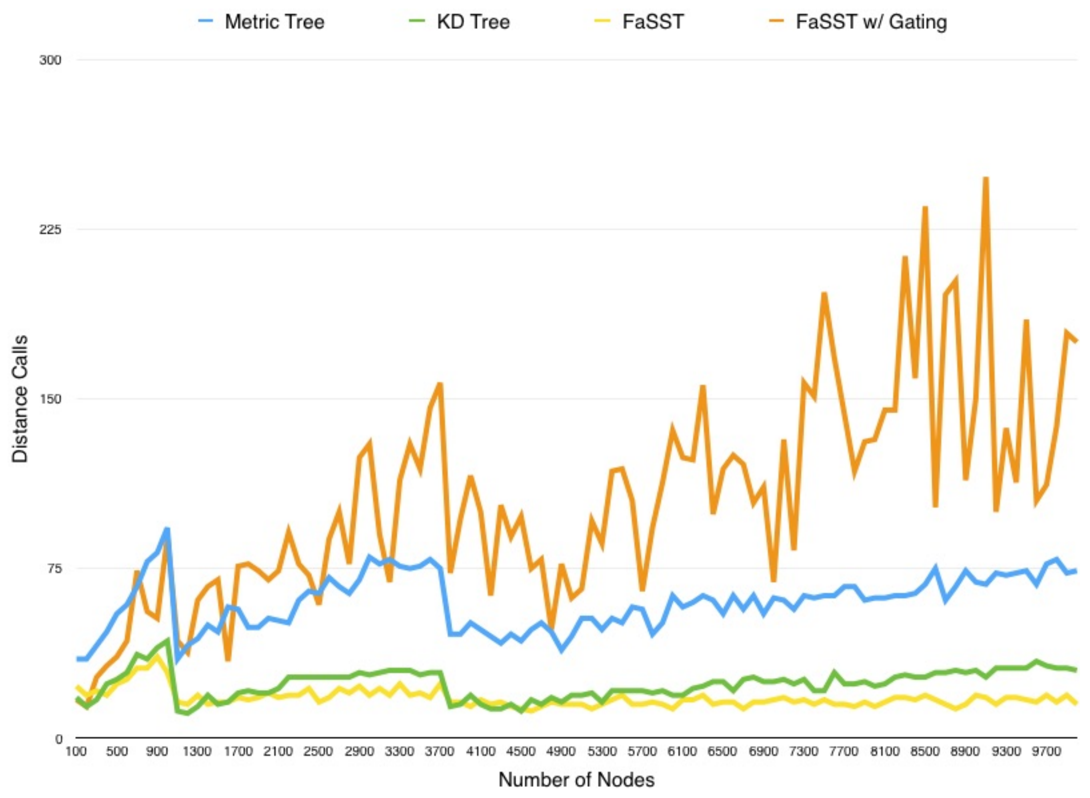

Figure 4.1: Euclidean Distance - 2d - Distance Calls

You will notice a large amount of variation with the FaSSTwith Gating. In low dimensions, the efficacy of of gating on the search is random because points tend to be close together. This *closeness* may lead to a very quick search but may also lead to more of the tree being explored resulting in many distance calls being performed.

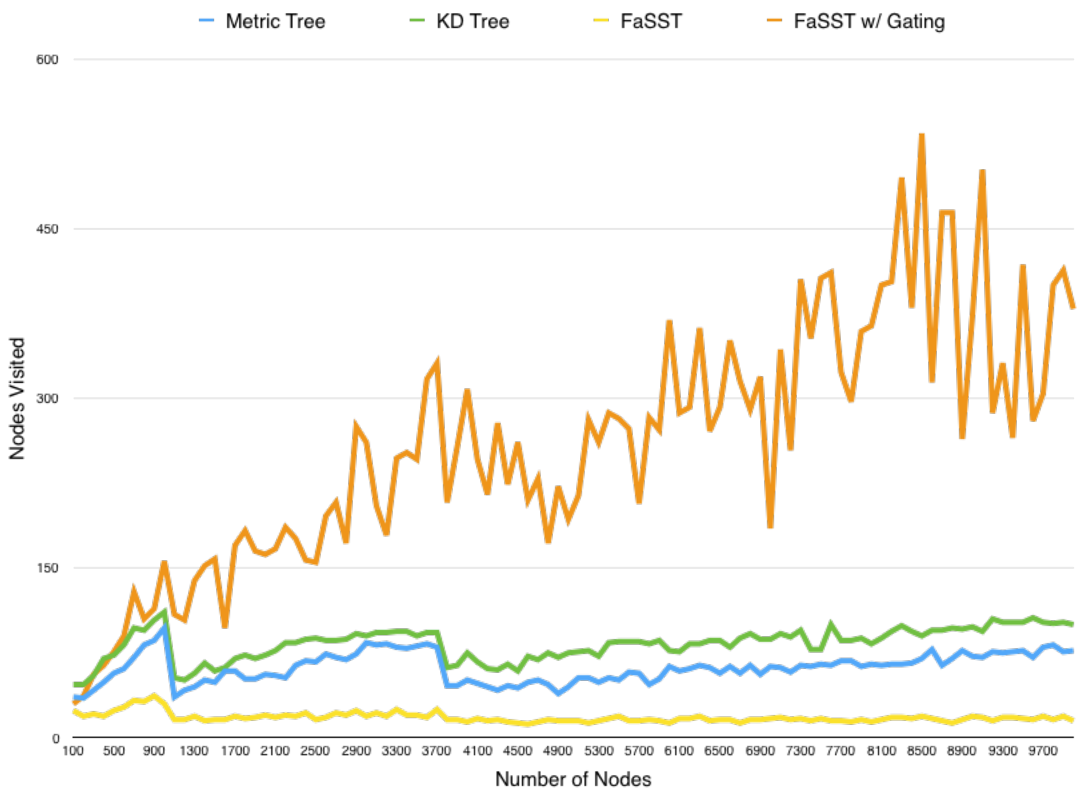

Figure 4.2: Euclidean Distance - 2d - Nodes Visited

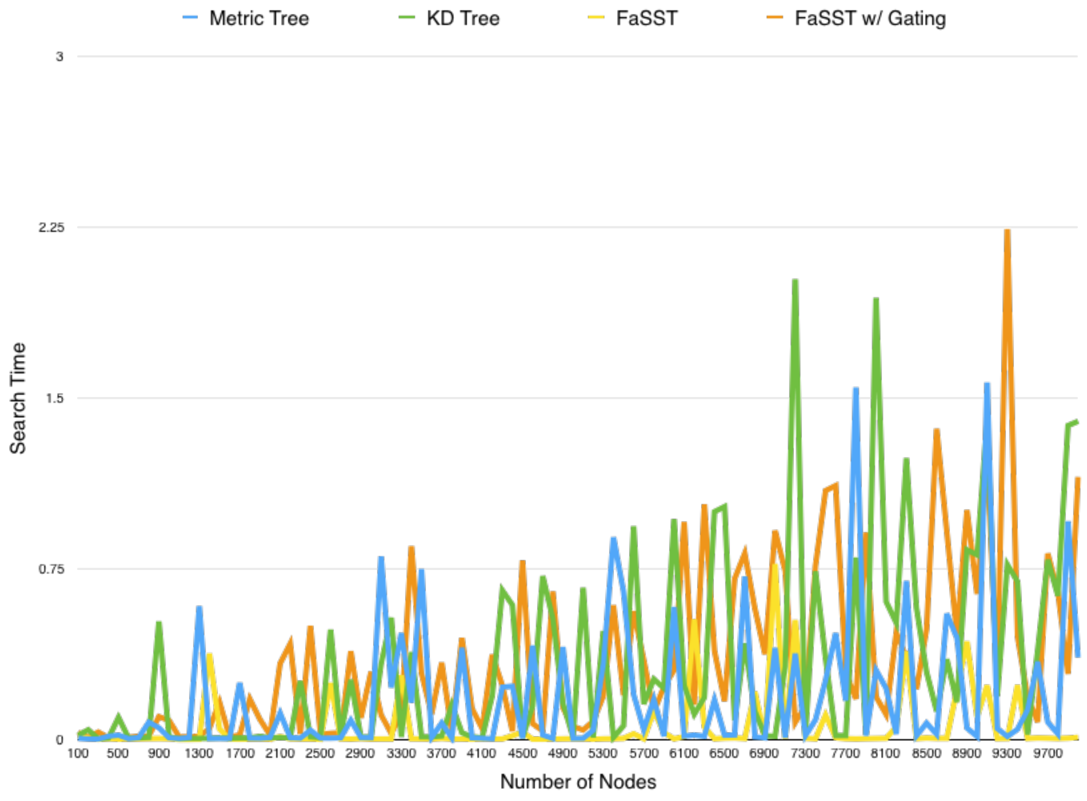

Figure 4.3: Euclidean Distance - 2d - Search Time

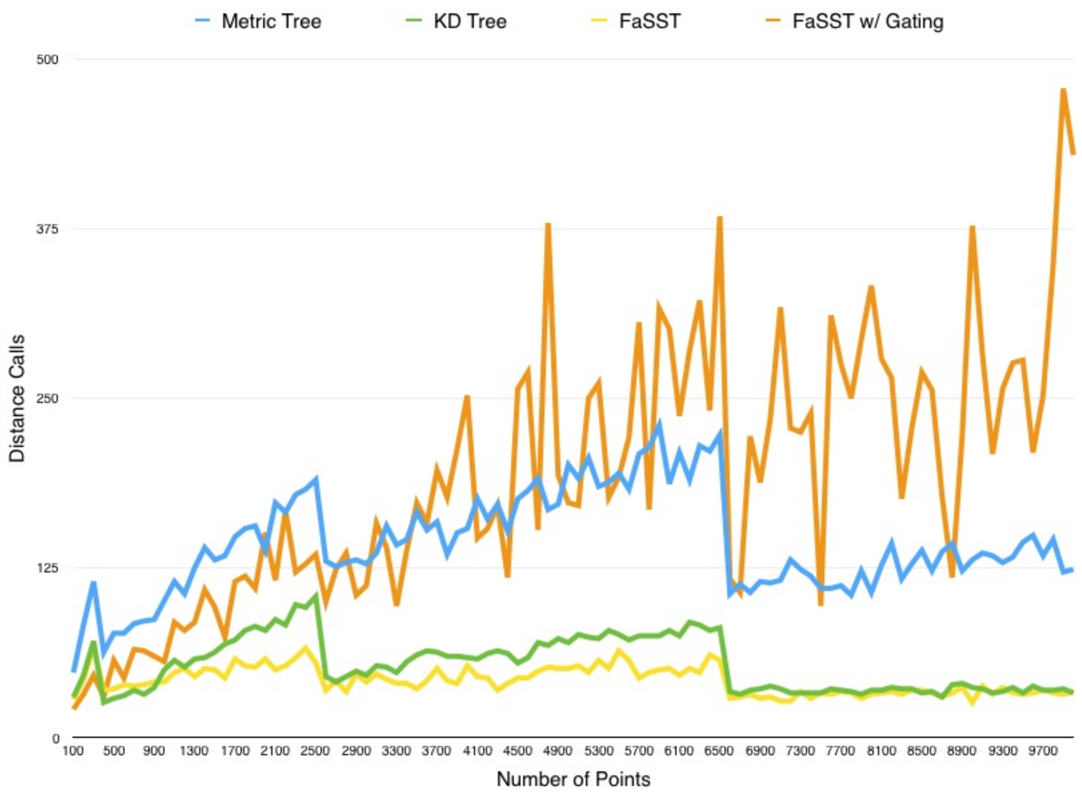

Figure 4.4: Euclidean Distance - 3d - Distance Calls

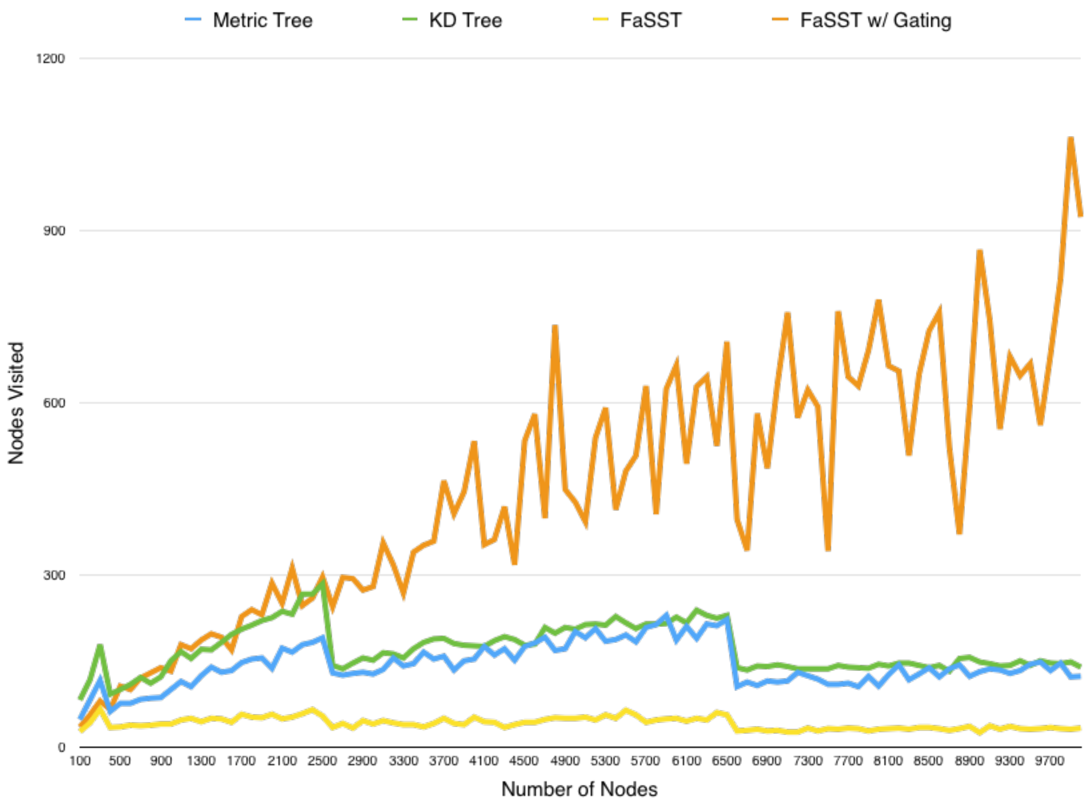

Figure 4.5: Euclidean Distance - 3d - Nodes Visited

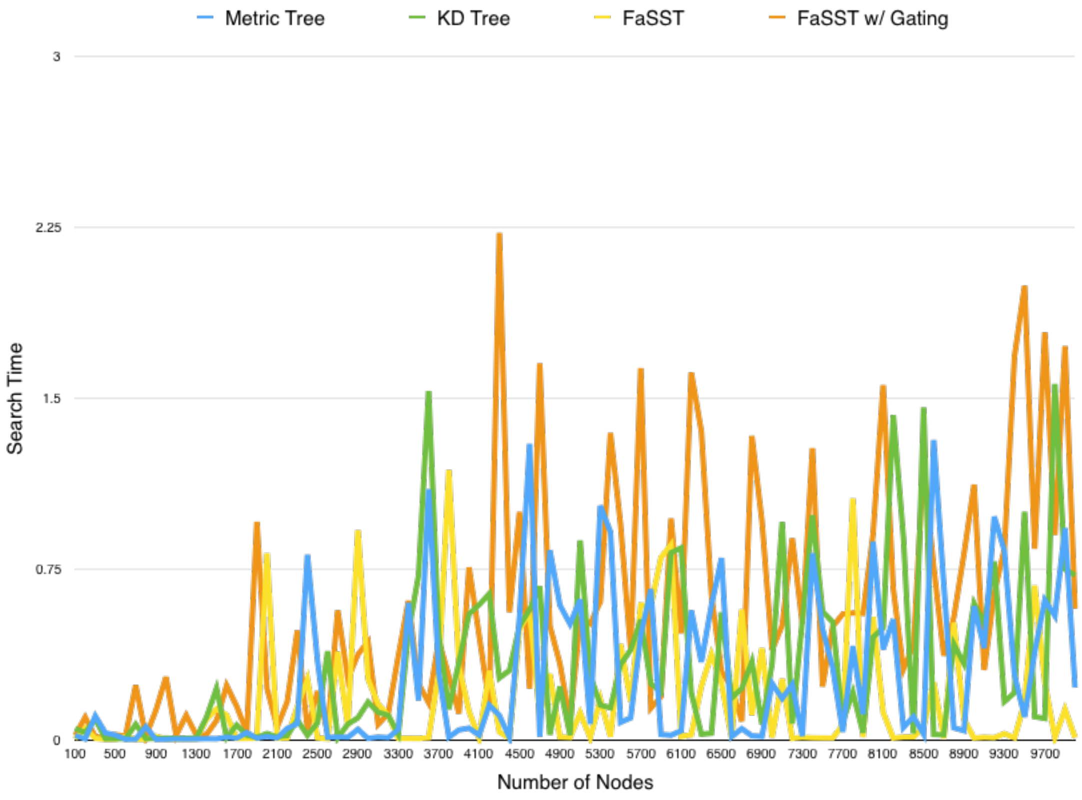

Figure 4.6: Euclidean Distance - 3d - Search Time

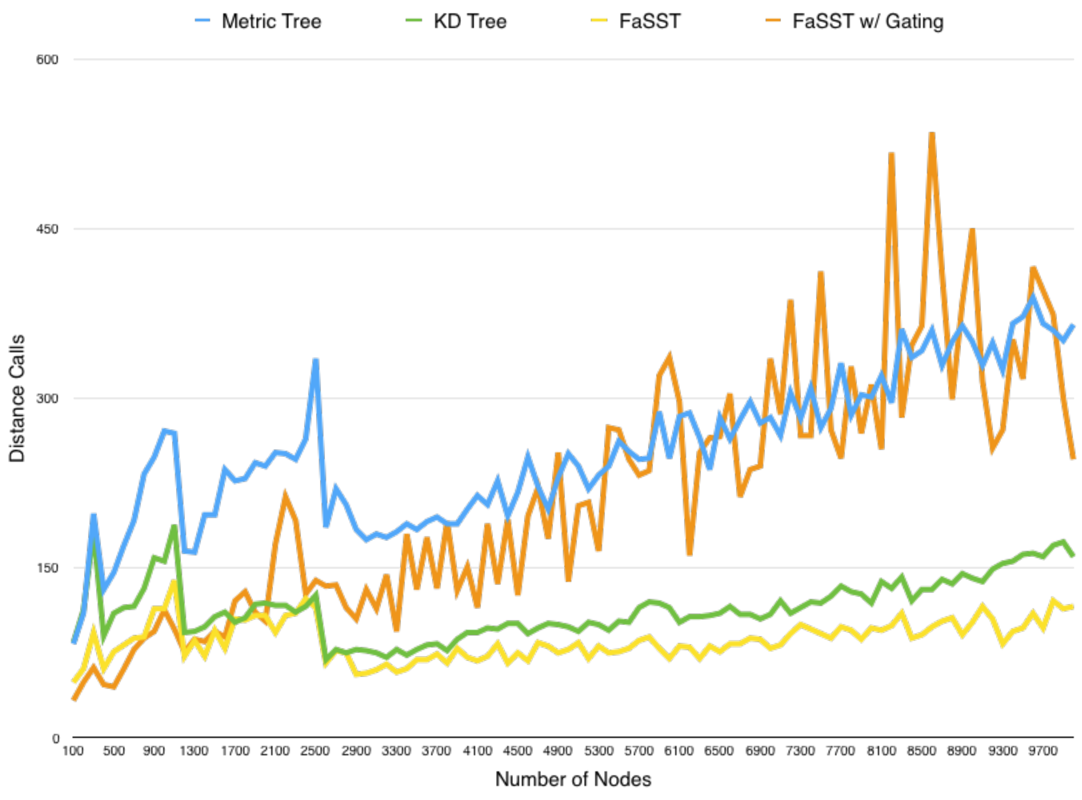

Figure 4.7: Euclidean Distance - 4d - Distance Calls

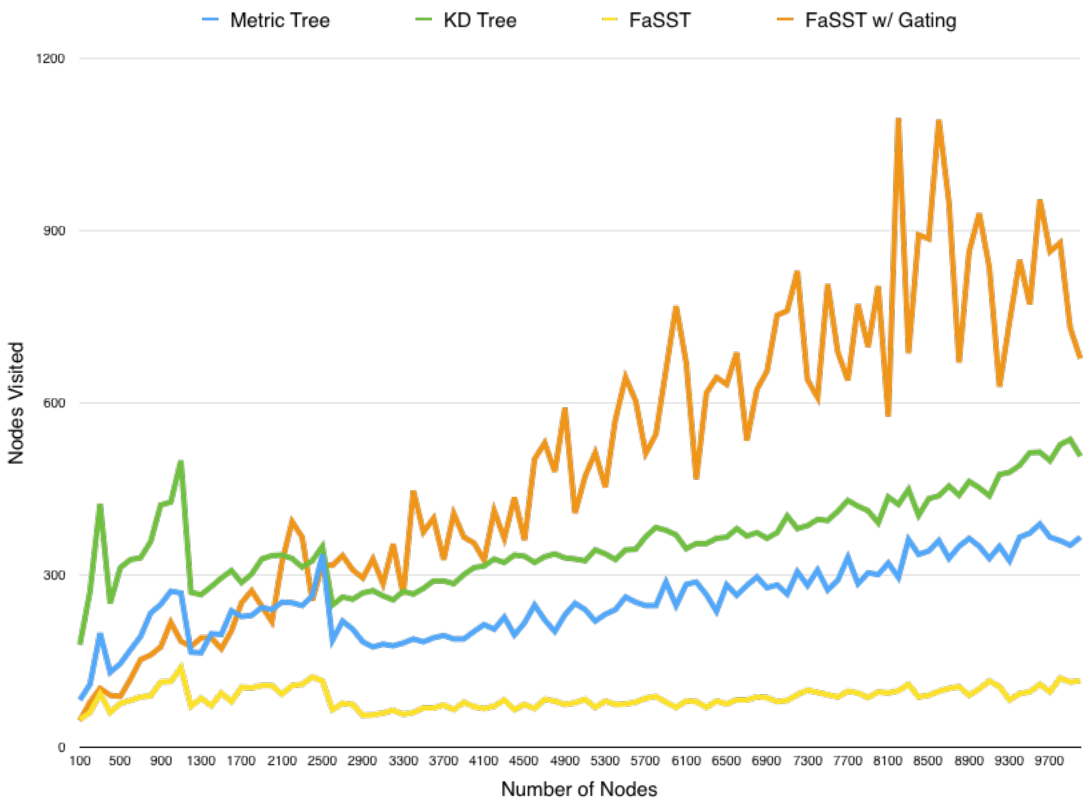

Figure 4.8: Euclidean Distance - 4d - Nodes Visited

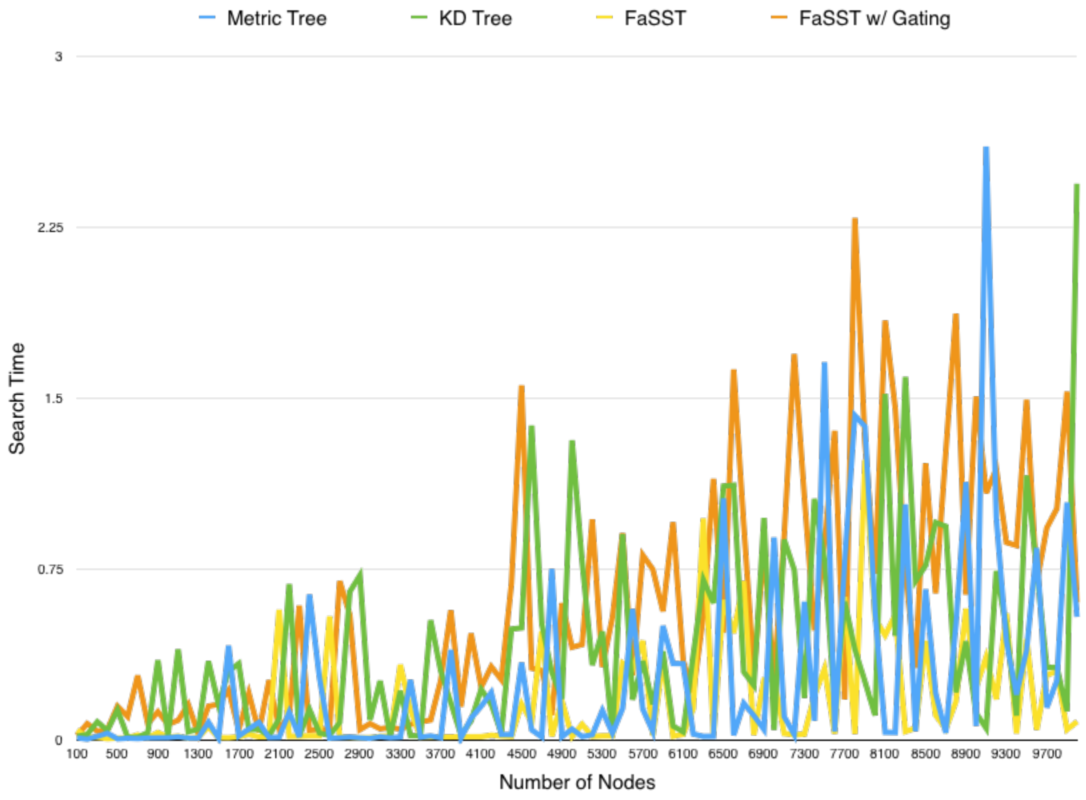

Figure 4.9: Euclidean Distance - 4d - Search Time

4.2.2 Dimensions 5 - 7

As the dimensionality increases the performance of the k-d tree and metric tree begin to degrade while the FaSST stays roughly constant. The FaSST with Gating begins to improve as it can make a more optimal balance of gating with pruning. While it is now making roughly the same number of distance calls as the FaSST, it tends to run a little slower because while the overhead of gating is minimal it is still present.

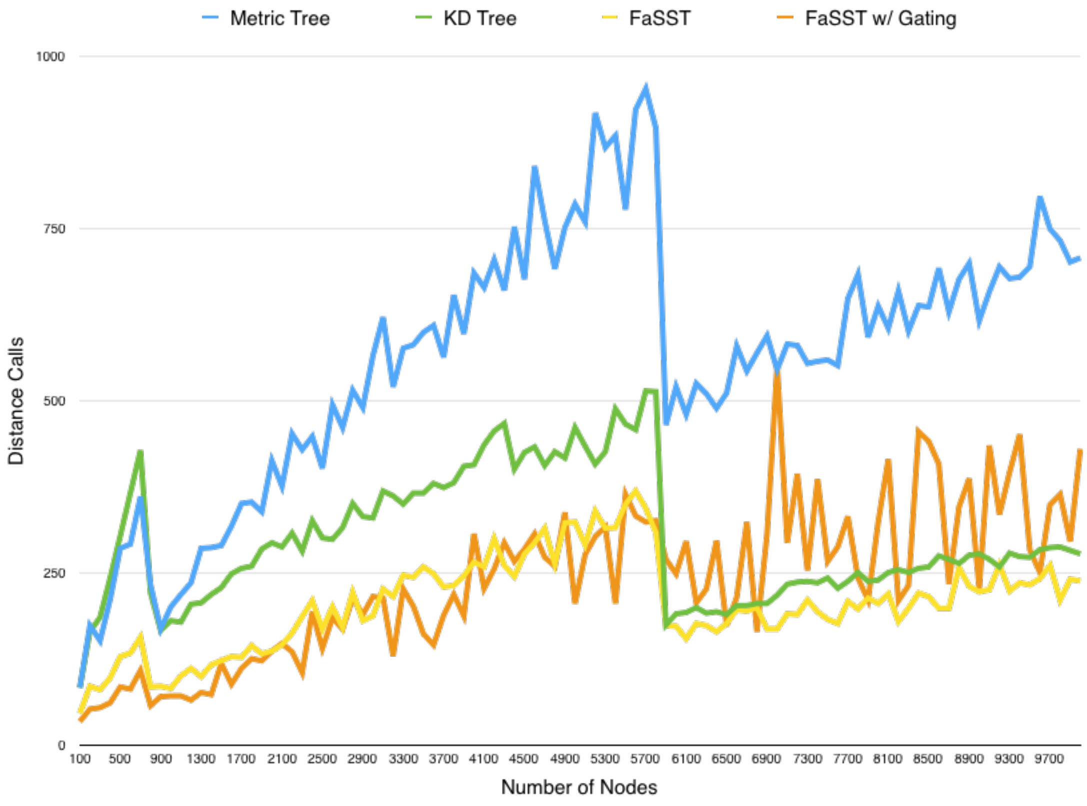

Figure 4.10: Euclidean Distance - 5d - Distance Calls

As the k-d tree begins to approach linear complexity certain drops begin to appear in the distance call and nodes visited graphs. This is because as the size of the tree grows, points become more clustered and so larger subtree's are able to be pruned while searching.

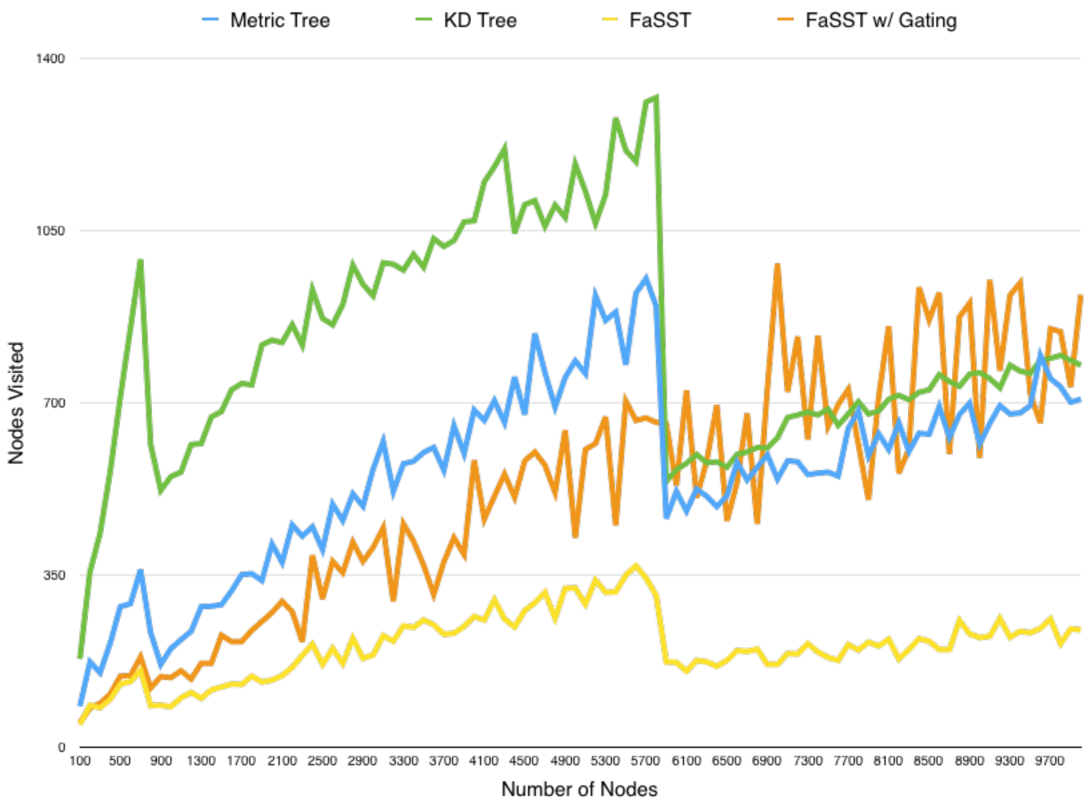

Figure 4.11: Euclidean Distance - 5d - Nodes Visited

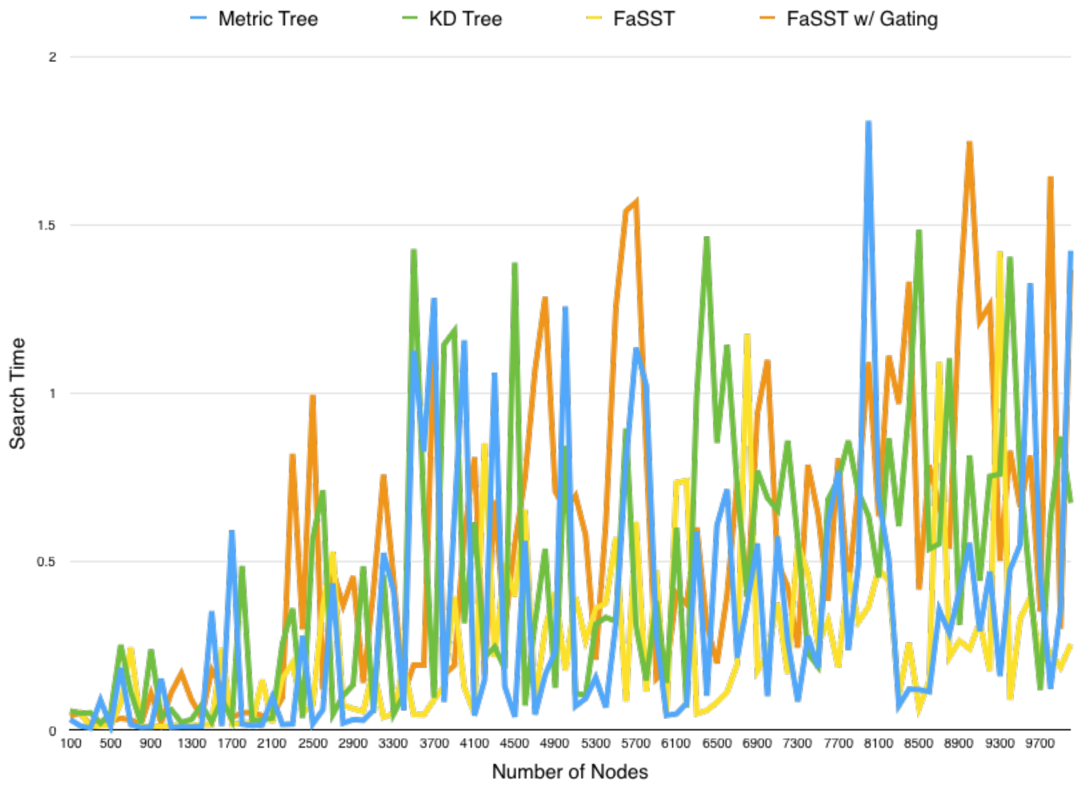

Figure 4.12: Euclidean Distance - 5d - Search Time

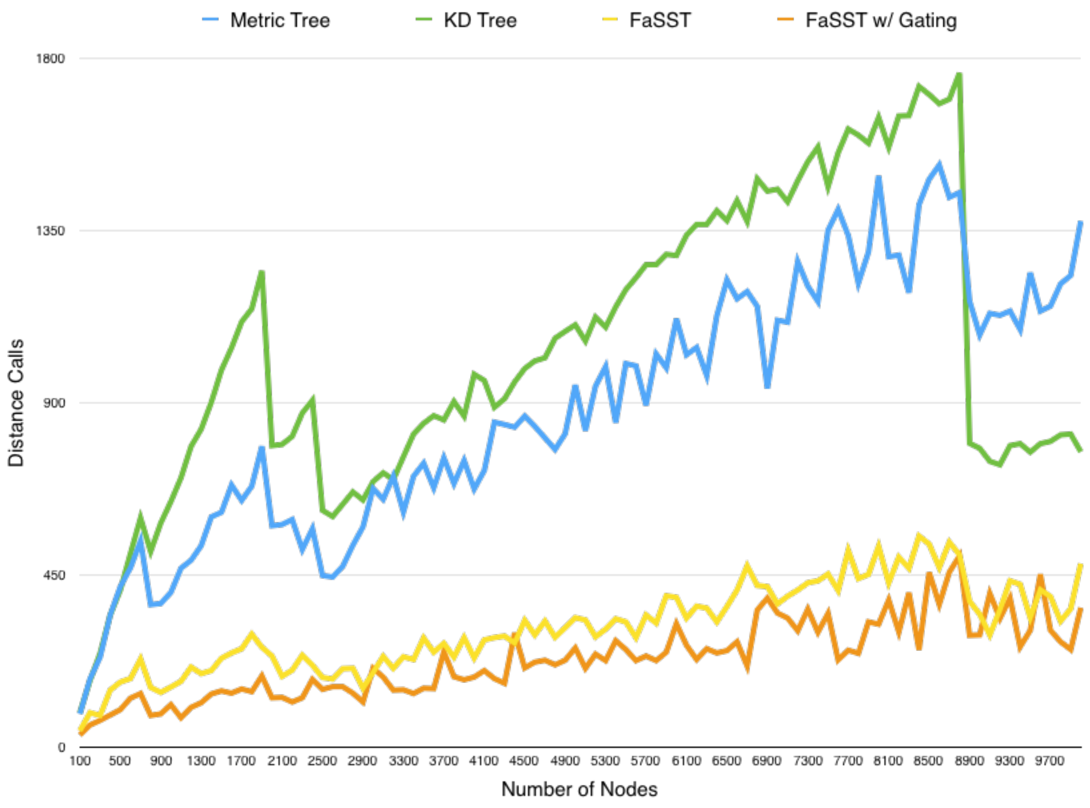

Figure 4.13: Euclidean Distance - 6d - Distance Calls

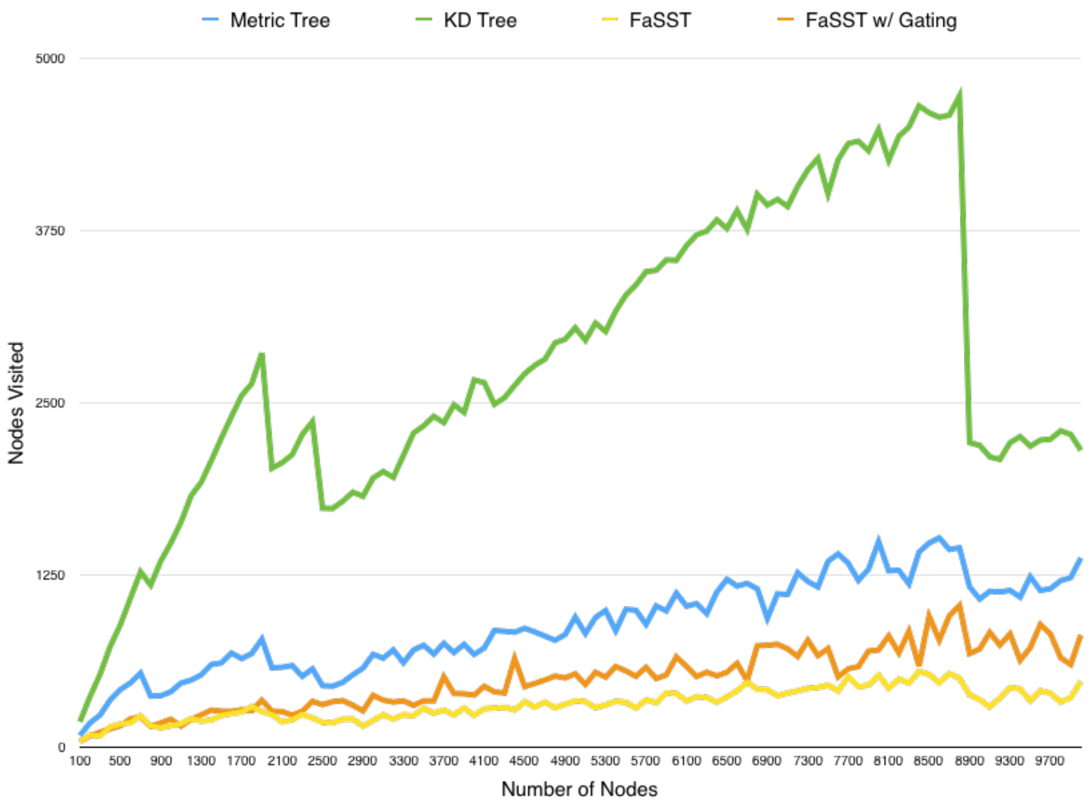

Figure 4.14: Euclidean Distance - 6d - Nodes Visited

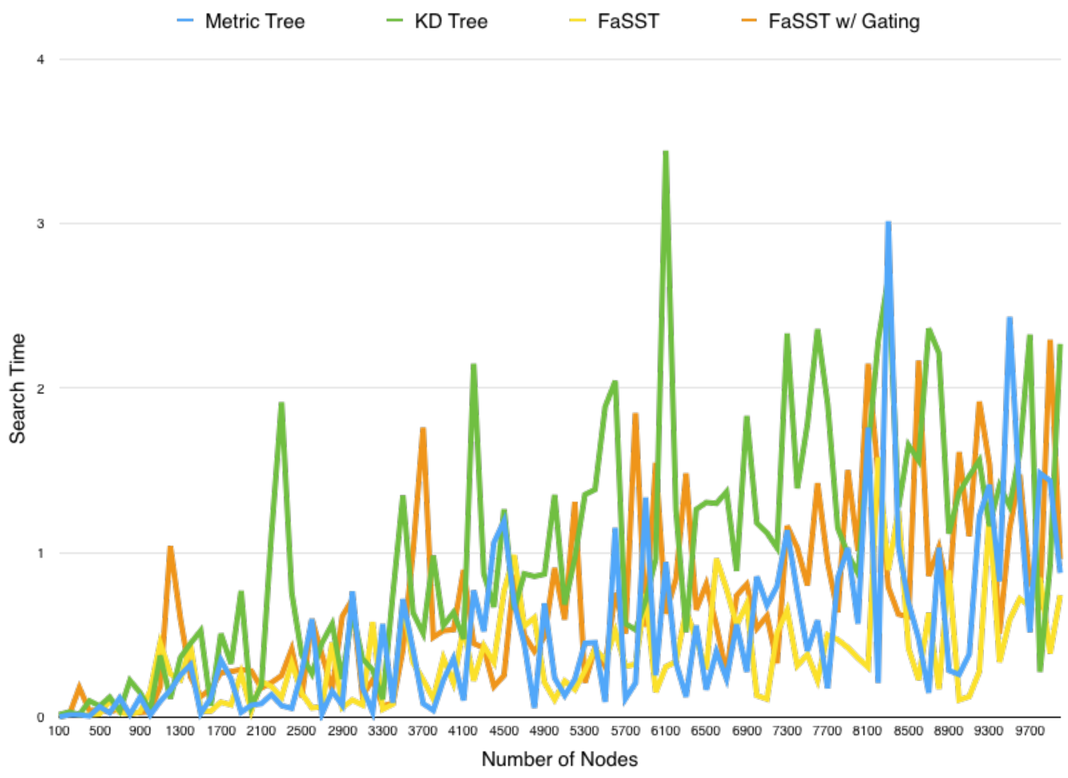

Figure 4.15: Euclidean Distance - 6d - Search Time

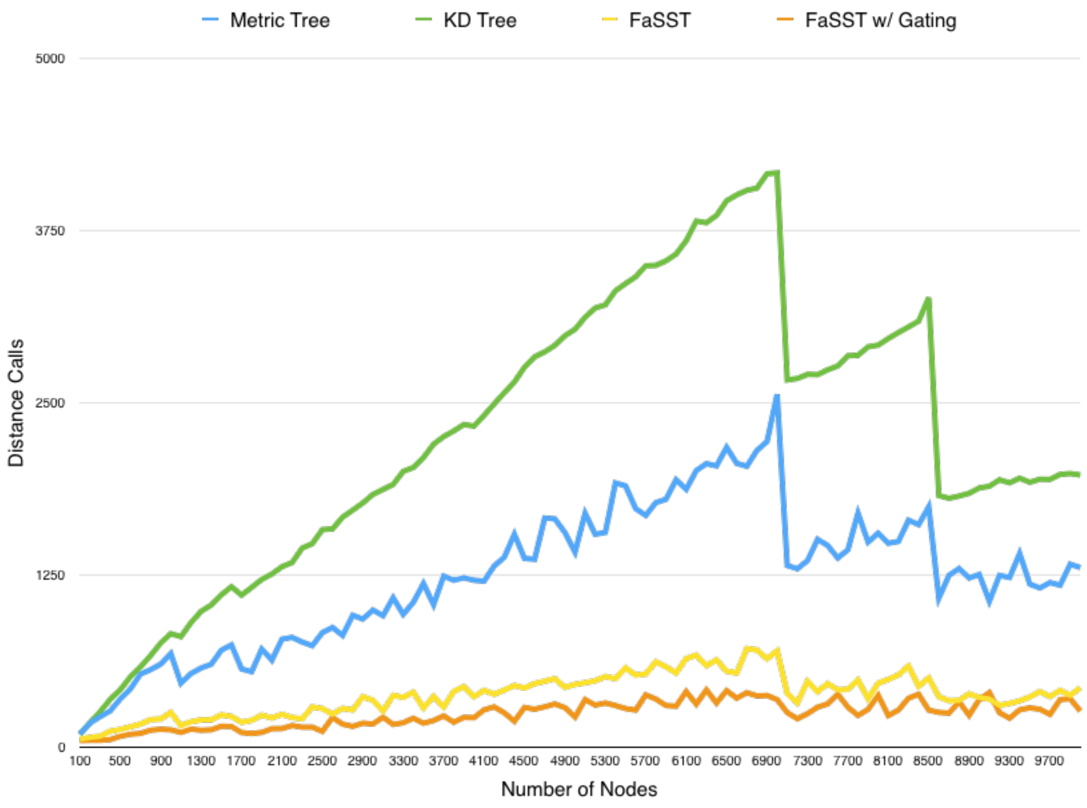

Figure 4.16: Euclidean Distance - 7d - Distance Calls

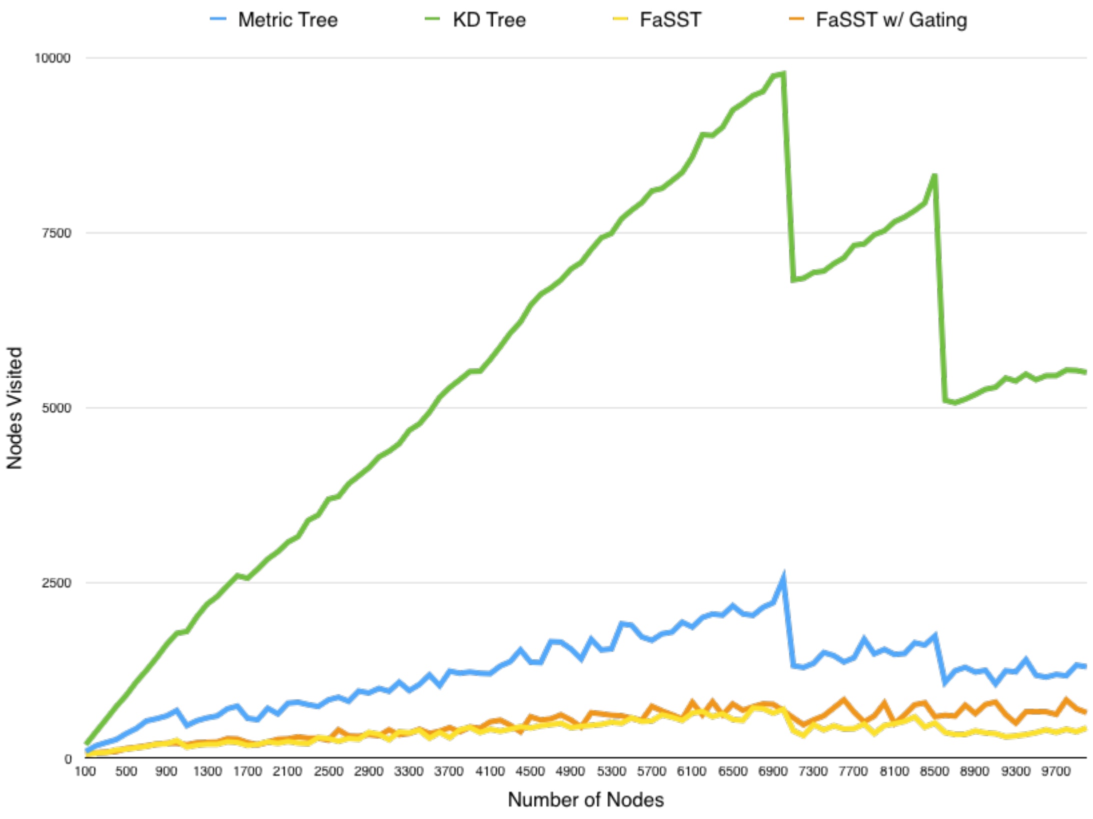

Figure 4.17: Euclidean Distance - 7d - Nodes Visited

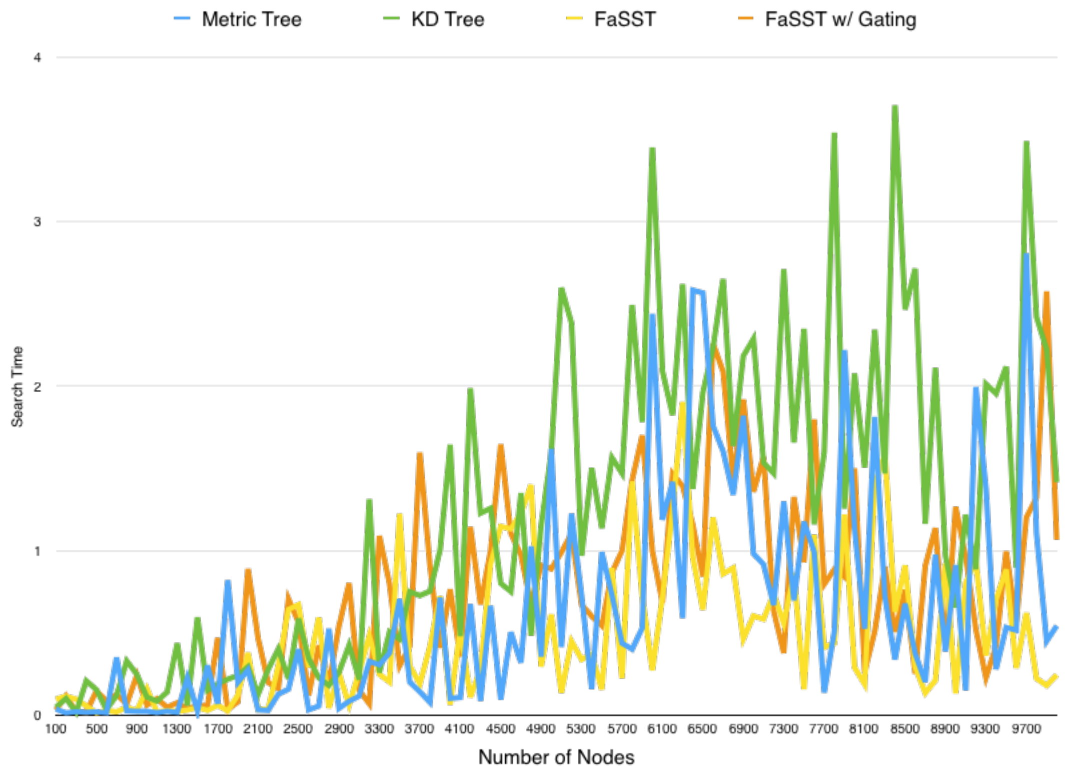

Figure 4.18: Euclidean Distance - 7d - Search Time

4.2.3 Dimensions 7 - 10

Once dimensionality reaches 8 and up the performance of the k-d tree go to to near linear and the performance of the FaSST and FaSST with Gating are consistently better than the metric tree. At is best the FaSST is running 65 times faster than the k-d tree and making 16 times fewer distance calls. At the same time, it is running 5 times faster than the metric tree and making 9 times fewer distance calls.

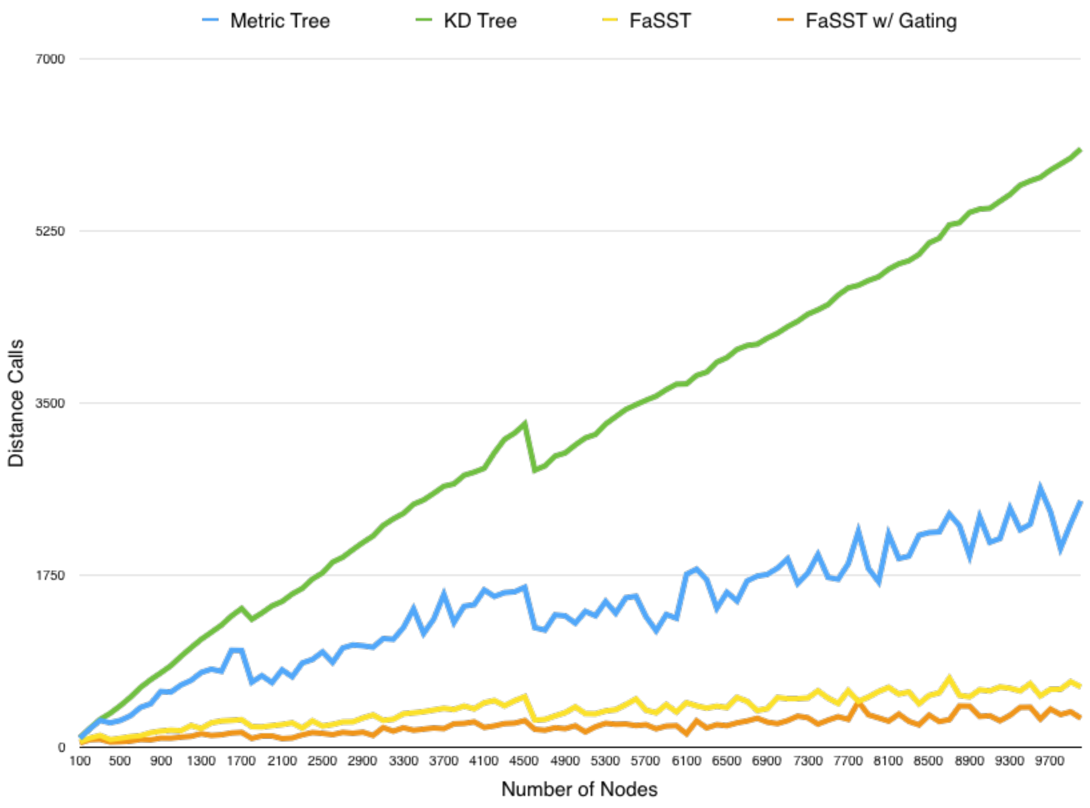

Figure 4.19: Euclidean Distance - 8d - Distance Calls

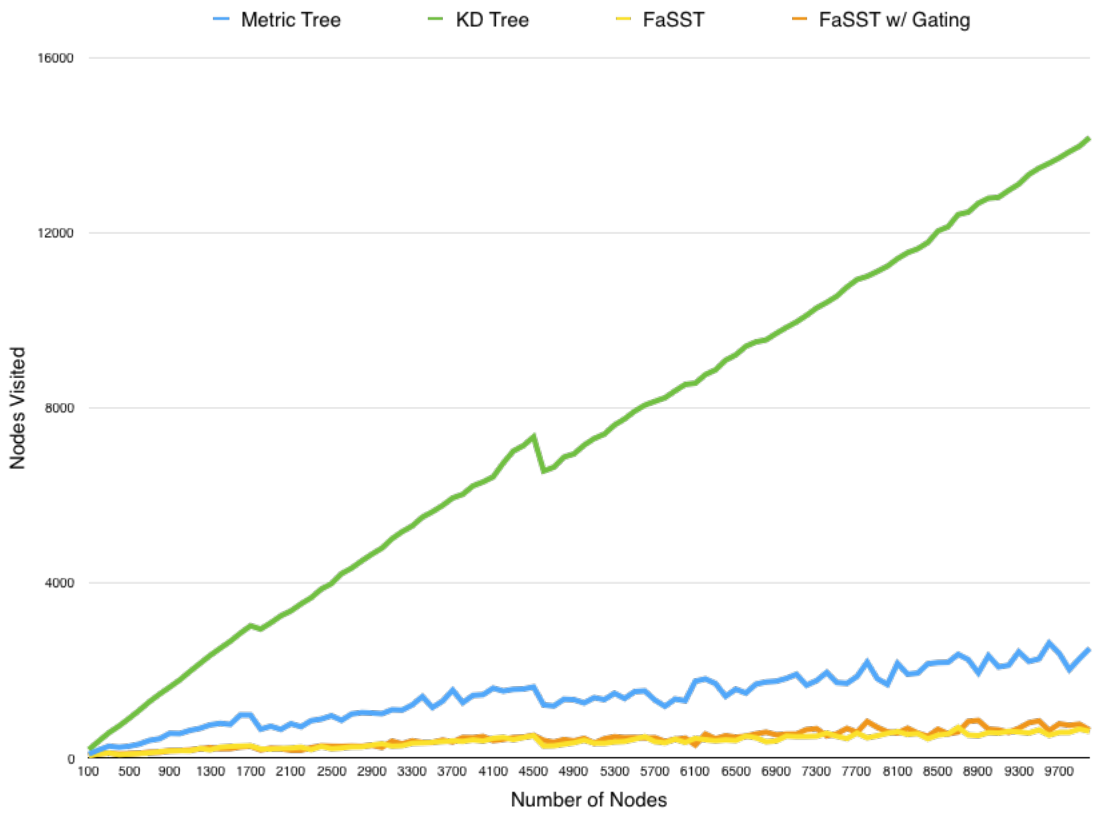

Figure 4.20: Euclidean Distance - 8d - Nodes Visited

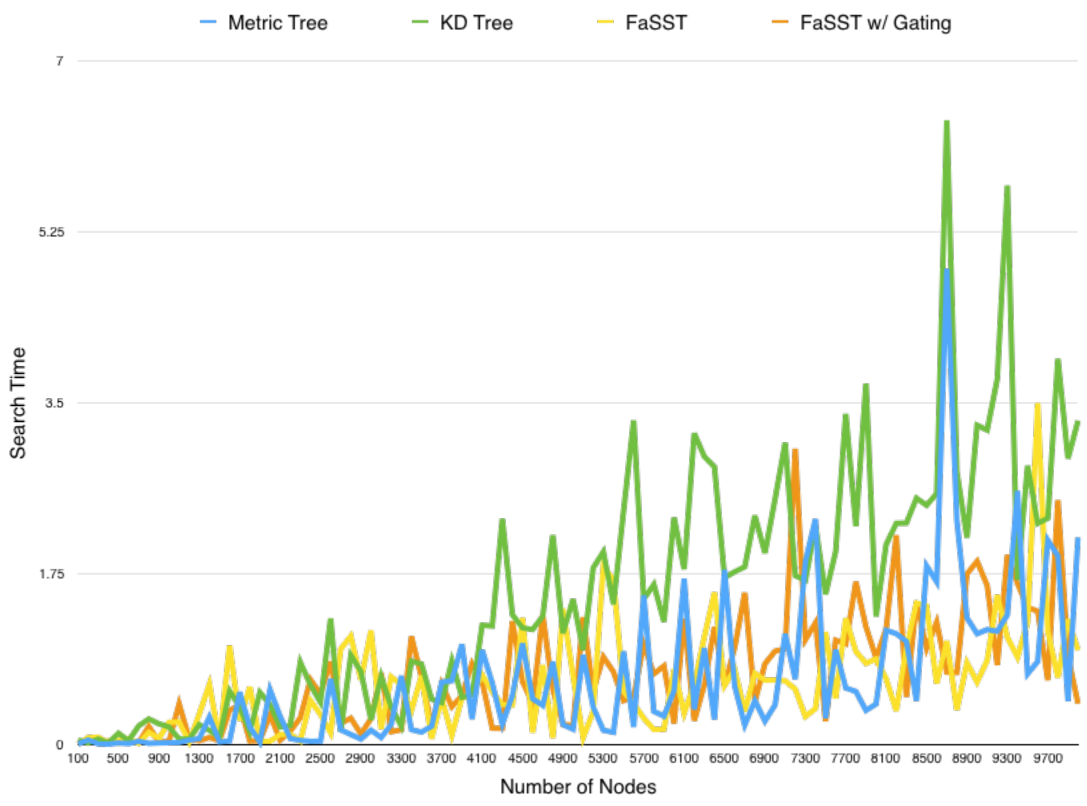

Figure 4.21: Euclidean Distance - 8d - Search Time

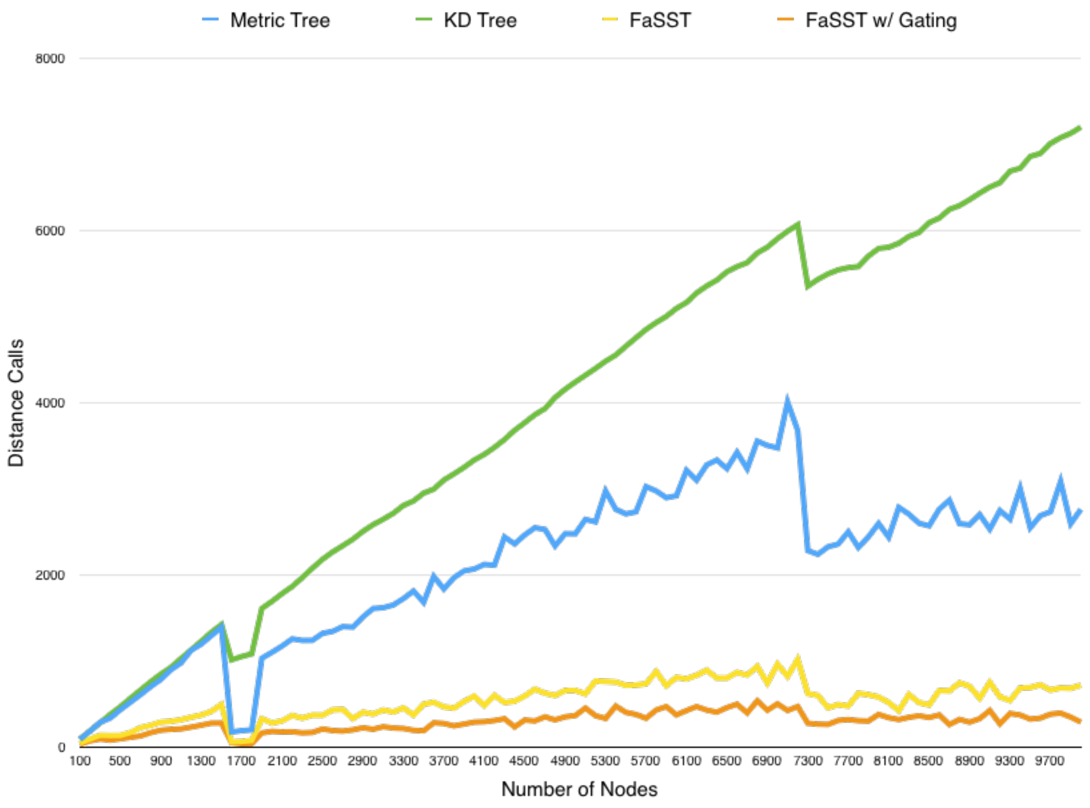

Figure 4.22: Euclidean Distance - 9d - Distance Calls

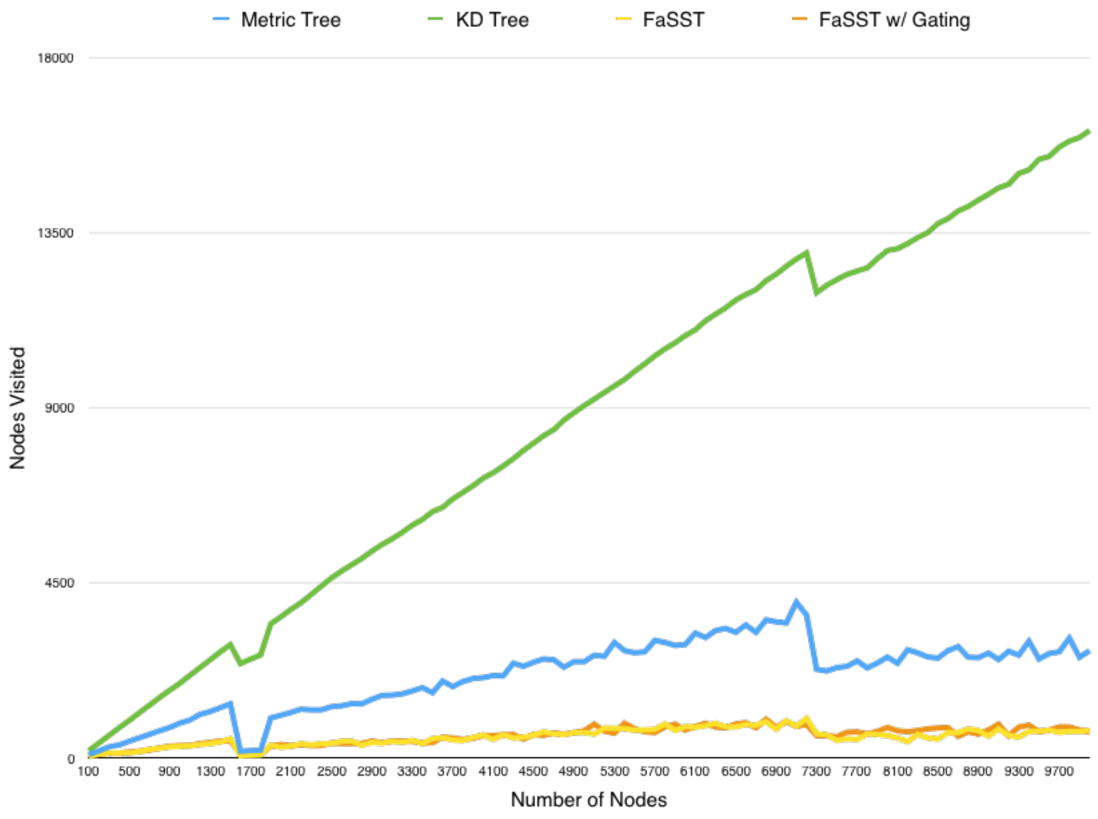

Figure 4.23: Euclidean Distance - 9d - Nodes Visited

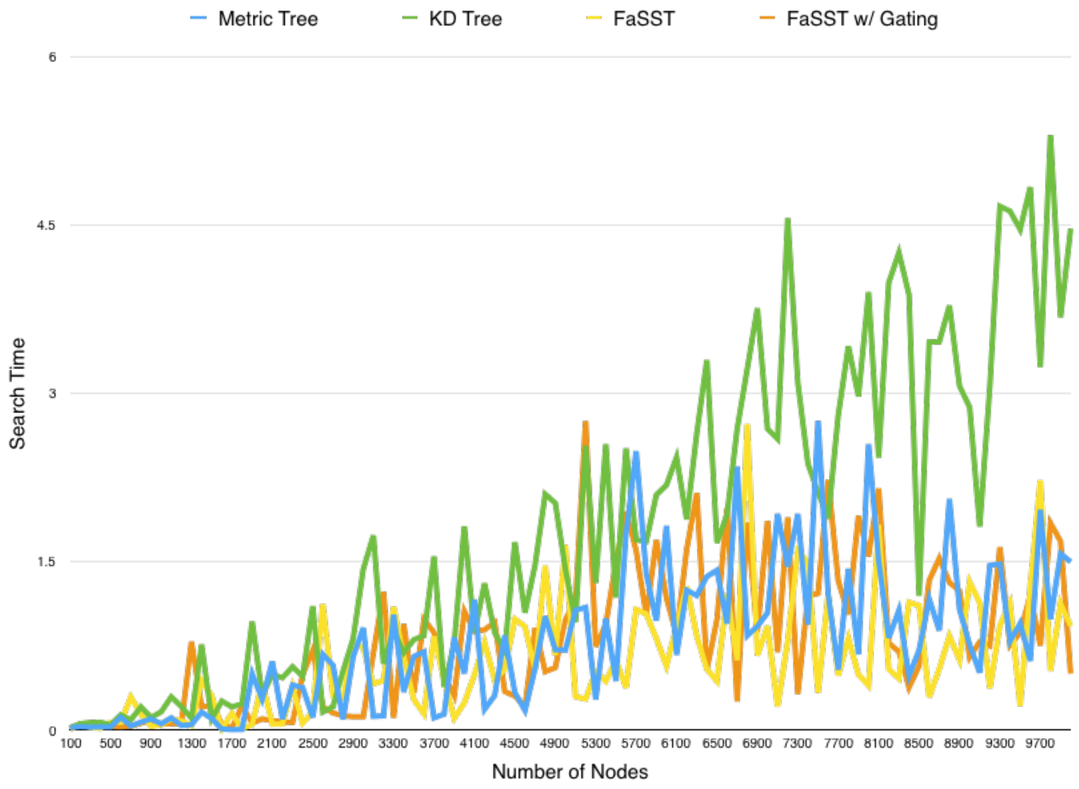

Figure 4.24: Euclidean Distance - 9d - Search Time

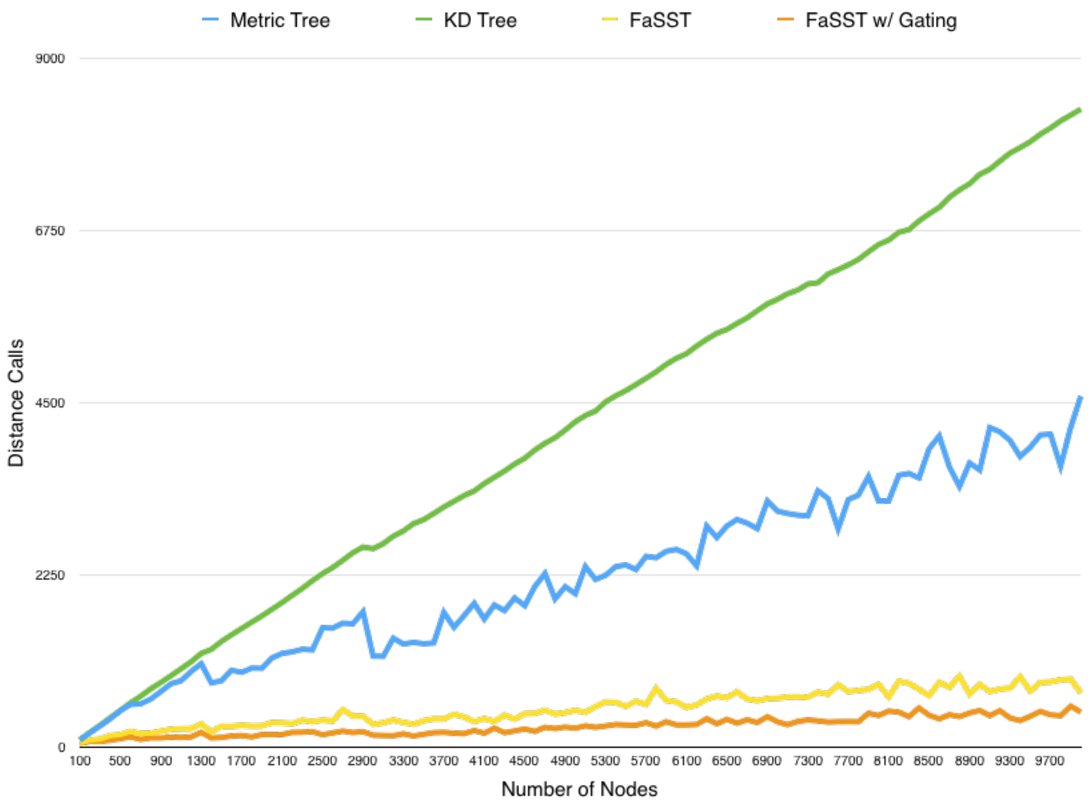

Figure 4.25: Euclidean Distance - 10d - Distance Calls

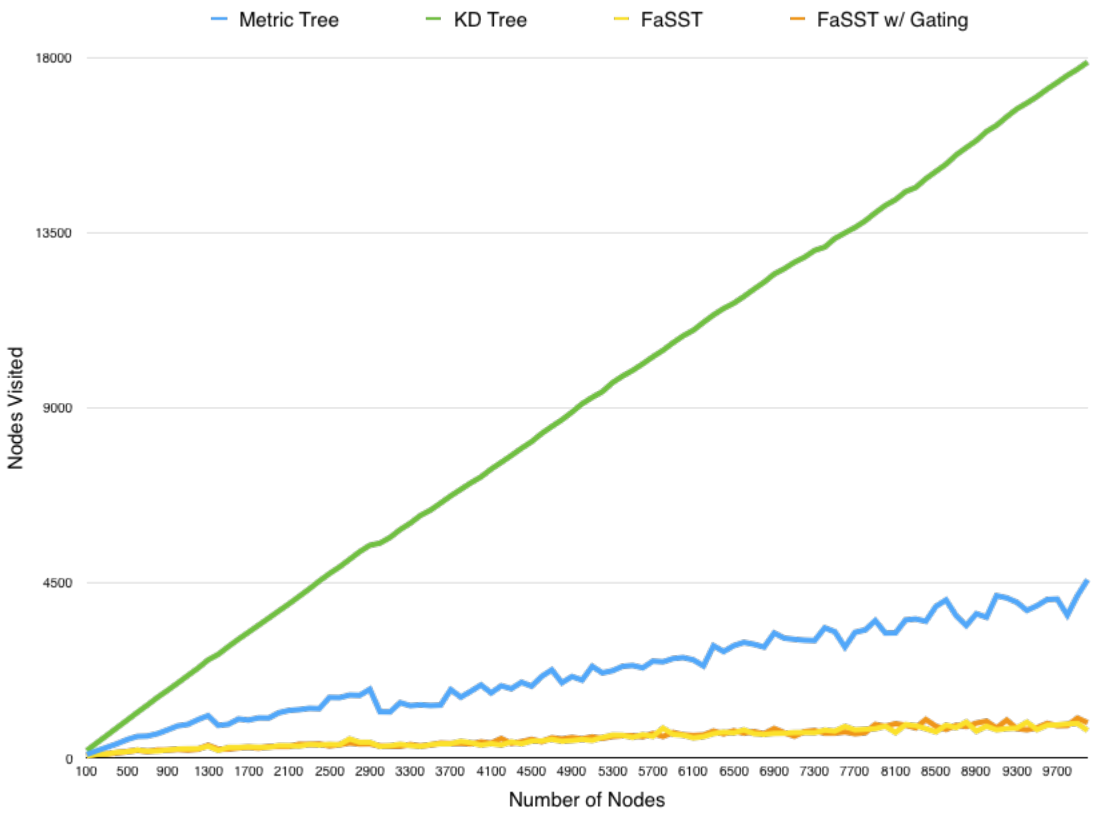

Figure 4.26: Euclidean Distance - 10d - Nodes Visited

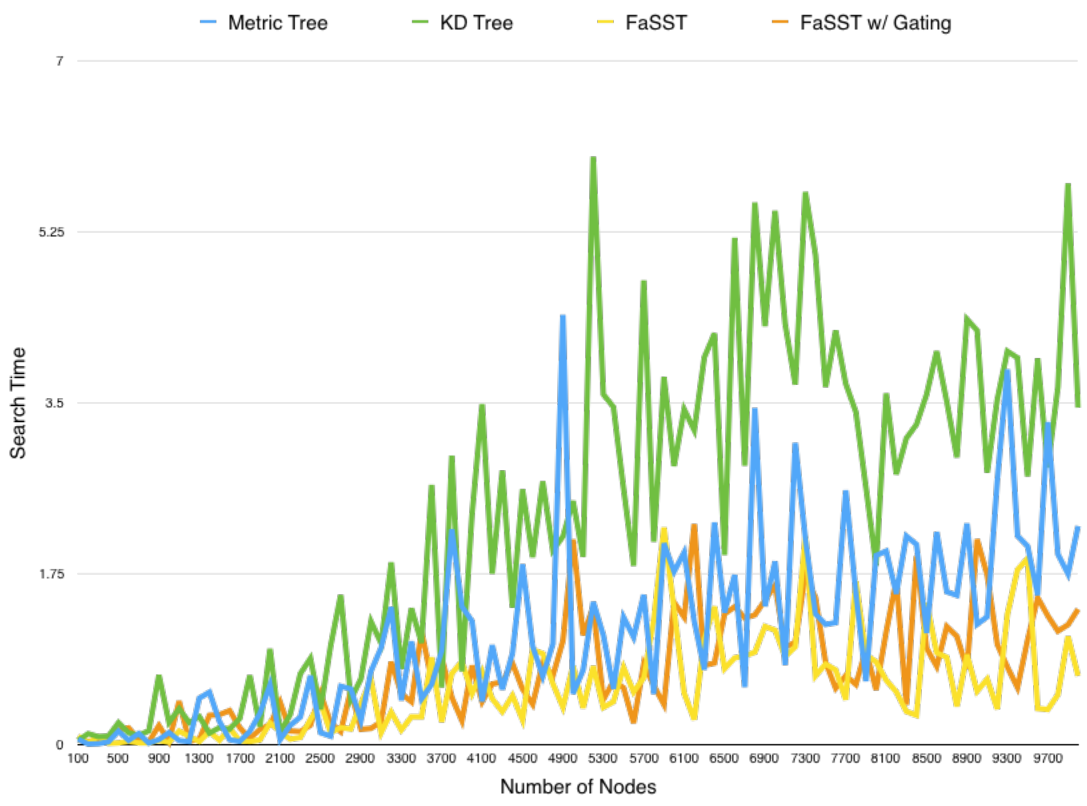

Figure 4.27: Euclidean Distance - 10d - Search Time

4.3 Non-Vector Spaces

This next section will examine the FaSST's performance in general metric spaces. Edit and hamming distances will be explored. Because these are non-vector spaces the k-d tree is no longer applicable and so it was not used in the following tests.

4.3.1 Edit Distance

As mentioned in chapter 2, edit distance measures the distance between two strings. All three tree's were built with 354,985 distinct english words. The target string used was "*hello*" and the radius was increased from 1 up to 27, the maximum radius which returned every element in the tree.

For all three measurements FaSST with Gating performed the best with FaSST, while the metric tree consistently performed the worst. The reason that all three tree's make less distance calls as the radius increases is because as the radius increases the query space fully encloses more and more subtrees.

Comparing the tree's based on distance calls shows that FaSST and FaSST with Gating are 15 and 30 times better than the metric tree respectively. They both visited roughly 3 times fewer nodes and ran 1.6 to 2.2 times faster than the metric tree.

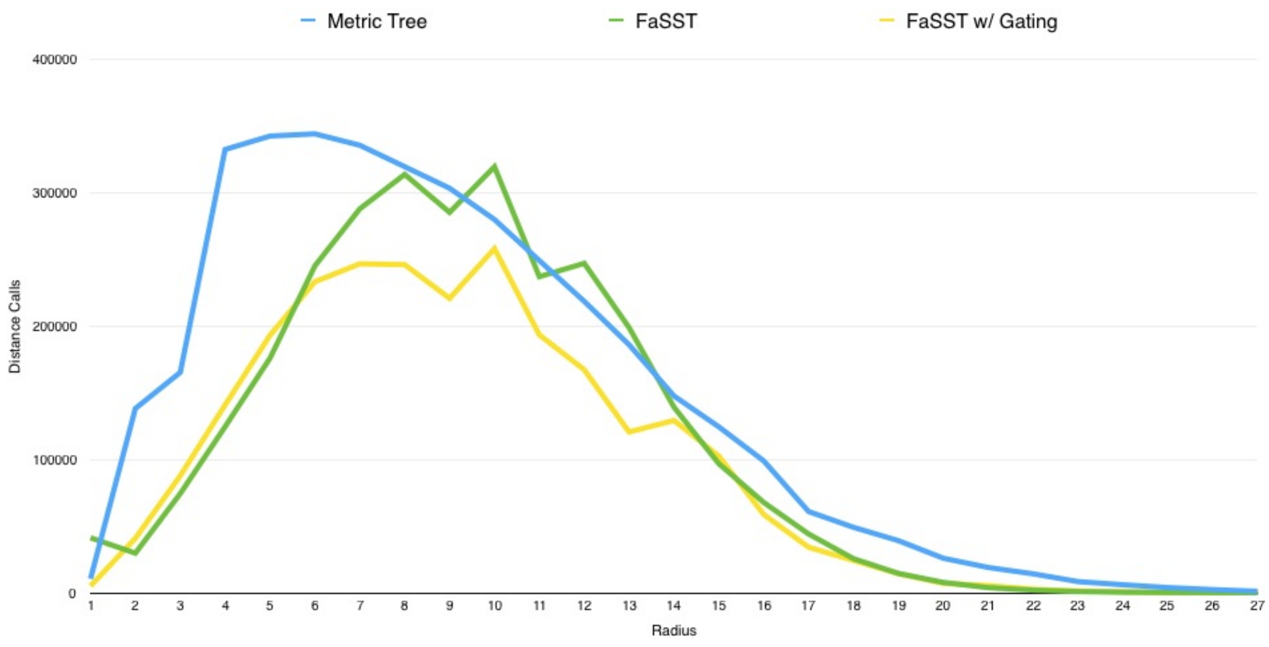

Figure 4.28: Edit Distance - Distance Calls

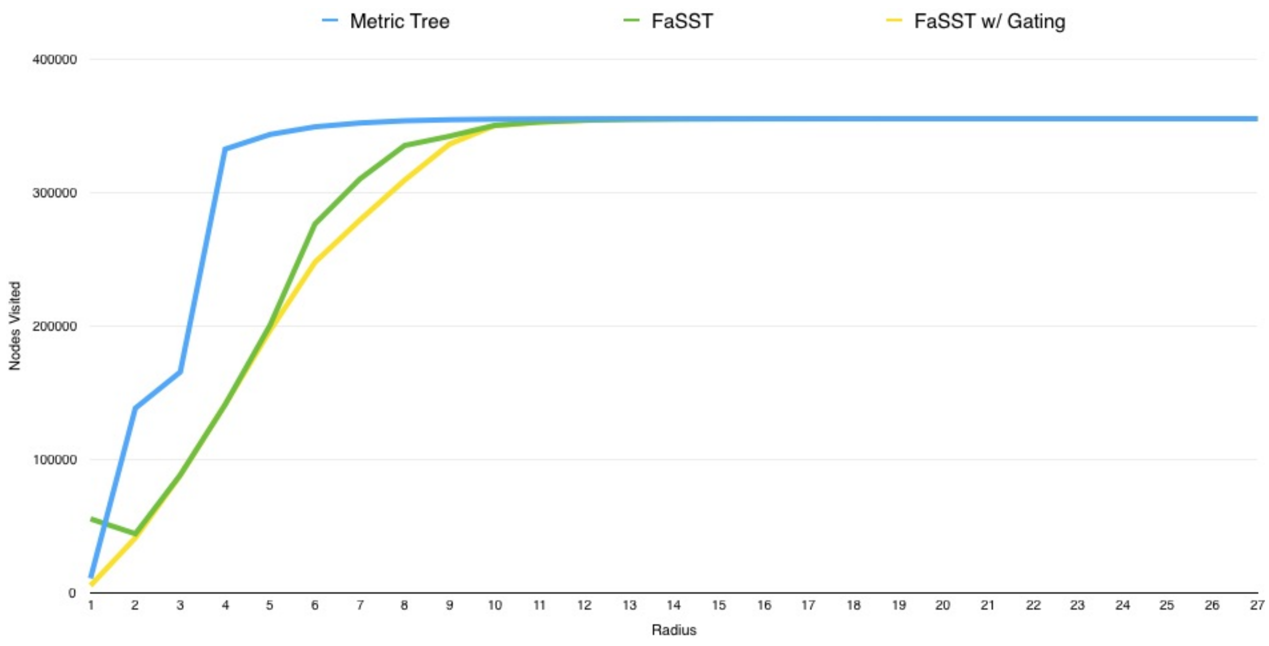

Figure 4.29: Edit Distance - Nodes Visited

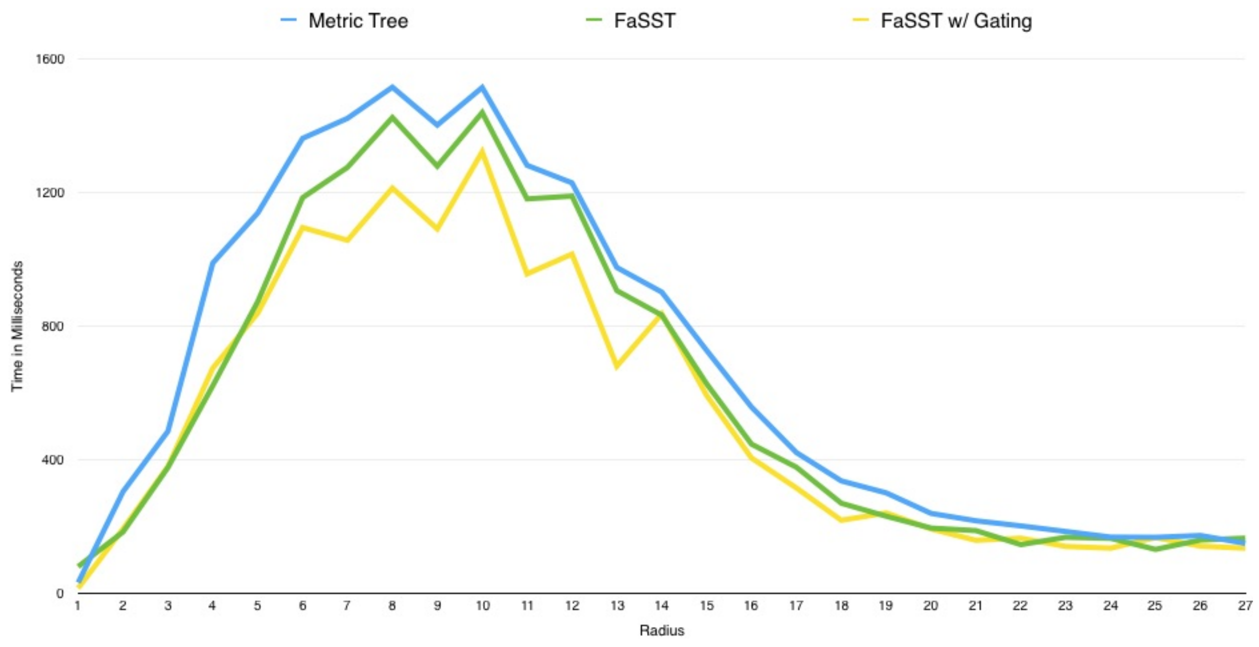

Figure 4.30: Edit Distance - Search Time

4.3.2 Hamming Distance

The tree's were also compared against hamming distance; the number of places two strings of equal length are different[26]. They were built will 500,000 distinct 32 bit integers and were queried based on the distance of the points from 0. The radius started at 1 and was increased until the max radius, where every point in the tree was returned.

The results are similar to the edit distance test; the FaSST with Gating performed the best with FaSST in second, while the metric tree performed the worst. Again, the reason that all three tree's make less distance calls as the radius increases is because as the radius increases the query space fully encloses more and more subtrees.

FaSST made up to 1.7 times fewer distance calls than the metric tree and FaSST with Gating made up to 16 times fewer calls. When comparing search times FaSST is 1.6 to 30.9 times faster than the metric tree and FaSST with Gating is 1.8 to 113.6 times faster. All tree's visited approximately the same number of nodes.

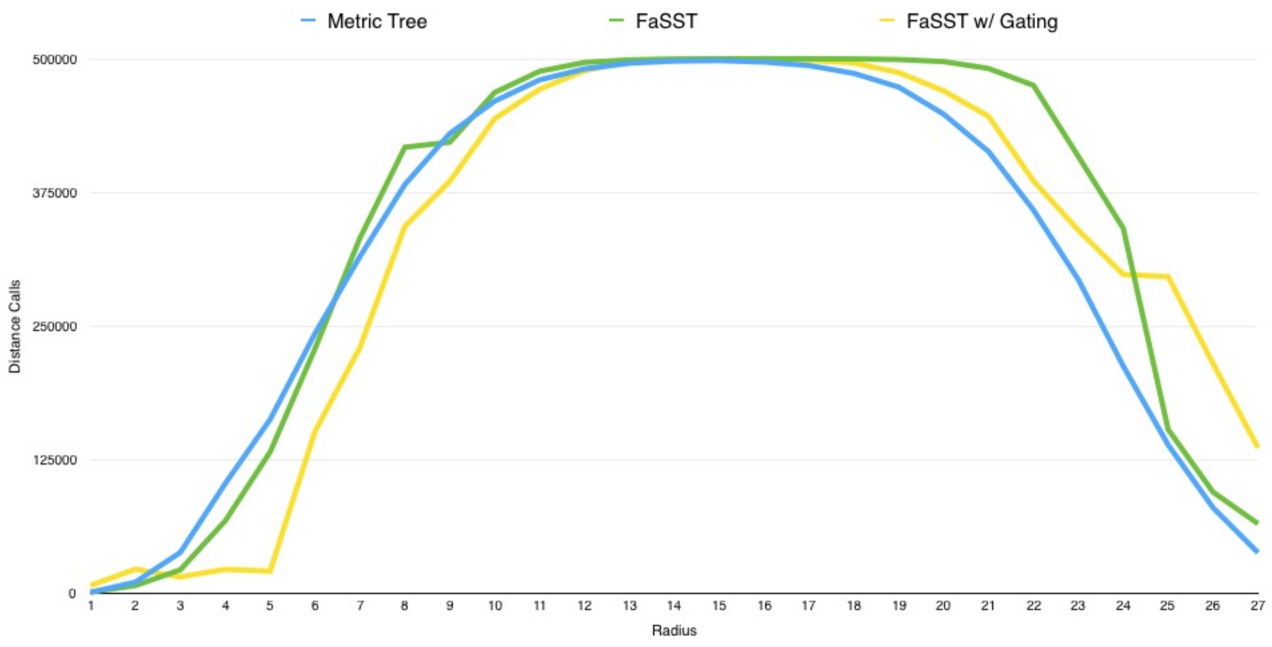

Figure 4.31: Hamming Distance - Distance Calls

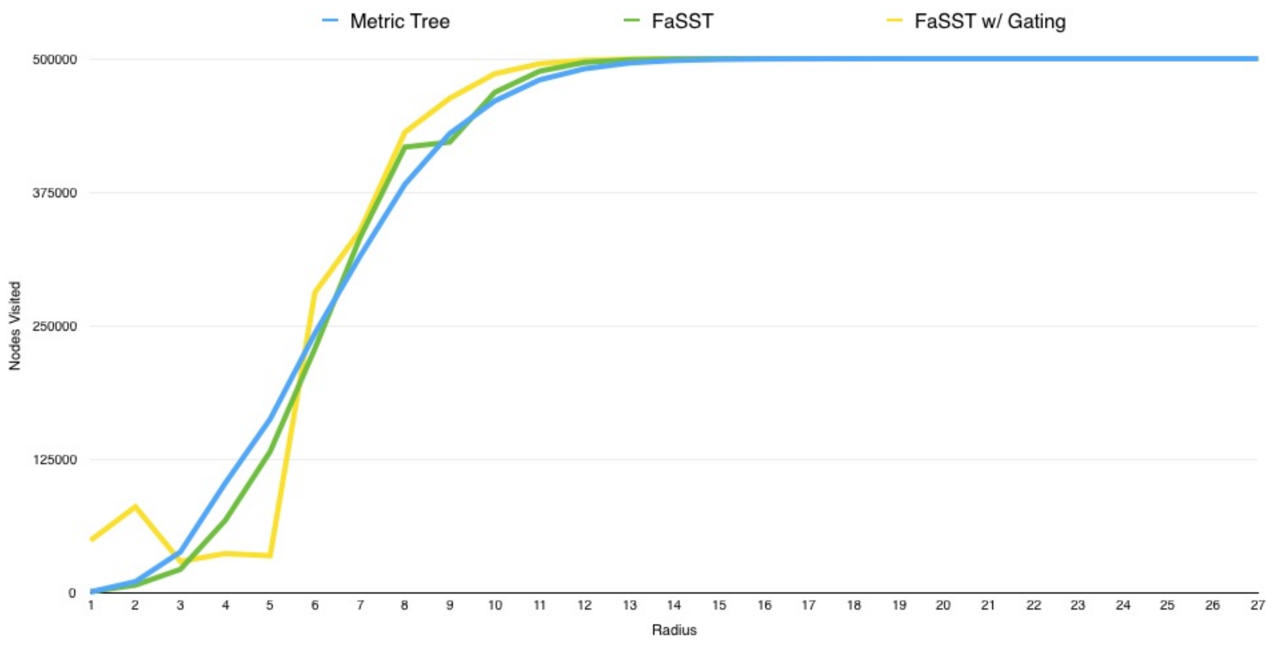

Figure 4.32: Hamming Distance - Nodes Visited

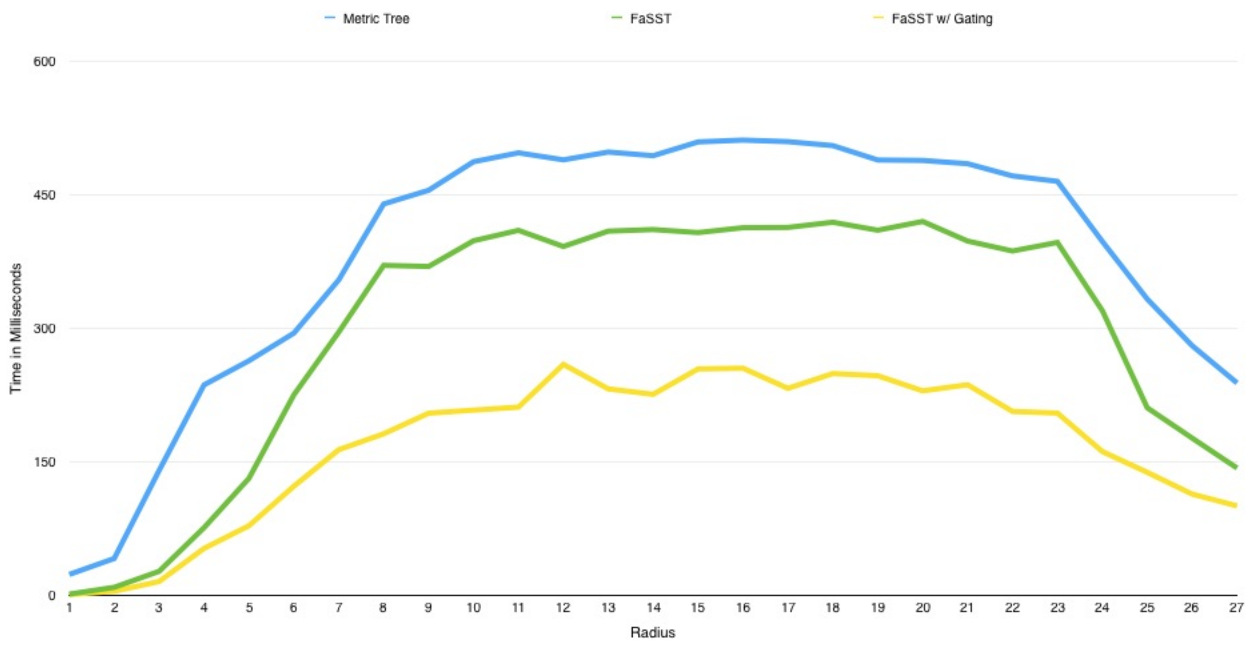

Figure 4.33: Hamming Distance - Search Time

4.4 Case Study: Smith-Waterman DNA Alignment

This chapter ends by presenting a real world example use of FaSST. The Smith-Waterman algorithm is a dynamic programming method for determining similarity between nucleotide or protein sequences. The algorithm was first proposed in 1981 by Smith and Waterman and is identifying homologous regions between sequences by searching for optimal local alignments. To find the optimal local alignment, a scoring system including a set of specified gap penalties is used[27]. Unlike edit distance, Smith-Waterman searches for local alignments, not global alignments, considering segments of all possible lengths to optimize the similarity measure.

The Smith-Waterman algorithm is the most accurate algorithm when it comes to search databases for sequence homology but it is also the most time consuming[28]. This makes it a perfect test candidate for this thesis.

Database similarity searches are among the best ways of gaining information about a given sequence, sequence comparisons are fundamental in bioinformatics[28]. The main reasons for performing database similarity searches between a nucleotide or protein query sequence of interest and sequences in a database include:

- Identify conserved domains in nucleotide or protein sequences of interest to predict functions of new and uncharacterized sequences
- Compare known sequences and identify homology between these sequences
- Compare sequences within taxonomic groups
- Search for a nucleotide sequence matching a protein sequence of interest or vice versa

- Search sequences in a database for patterns similar to the patterns in the sequence of interest

4.4.1 Experiment

The efficacy of FaSST and FaSST with Gating against the metric tree was tested on a dna dataset with the Smith-Waterman distance used as the metric. All three tree's were built with 1,614,950 dna sequences taken from Aligient Technologies online genomic data[29]. The tree's were queried with the same target point and gradually increasing search radius until all points in the data set were returned. For each test the number of distance calculations performed, nodes visited, and time take was recorded.

4.4.2 Smith-Waterman: Results

Until the limit both tree's perform much better than the metric tree with the FaSST making 13000 times fewer distance calls and FaSST with Gating making 26000 times fewer distance calls. Comparing just FaSST to FaSST with Gating we can see that Gating results in half the distance calls as opposed to no gating at all. The number of nodes visited follows a similar pattern except gating results in visiting a slightly larger number of nodes at the beginning. This is expected as gating causes more of the tree to be searched.

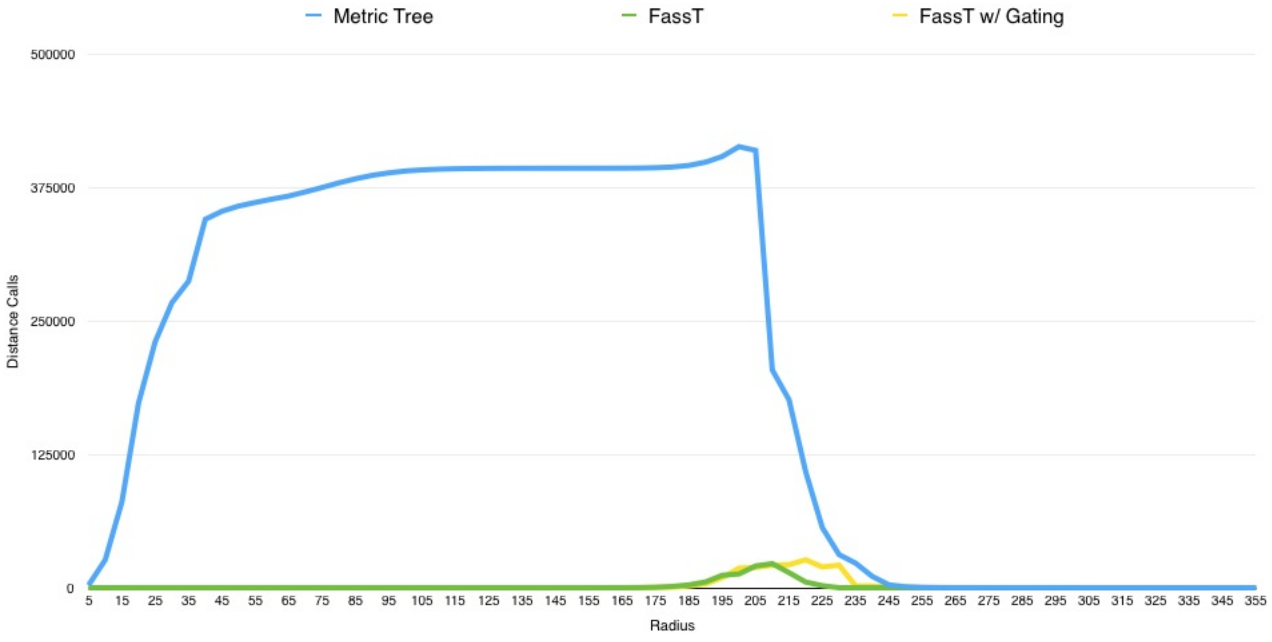

Figure 4.34: Smith-Waterman - Distance Calls

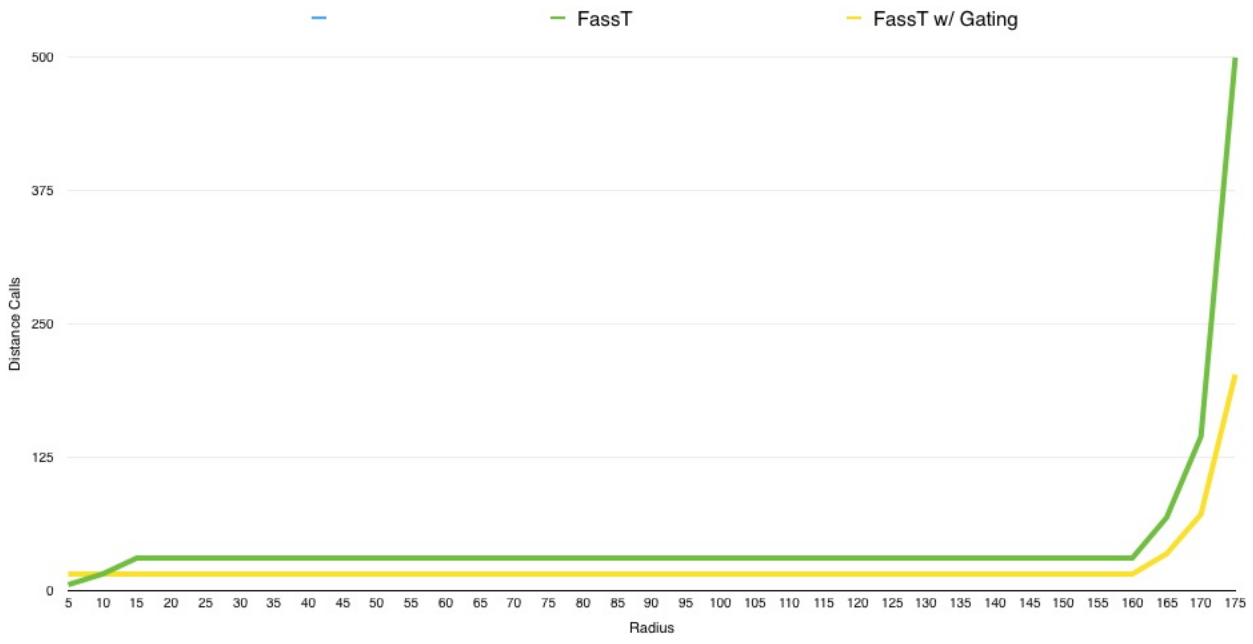

Figure 4.35: Smith-Waterman - Distance Calls - FaSST and FaSST with Gating

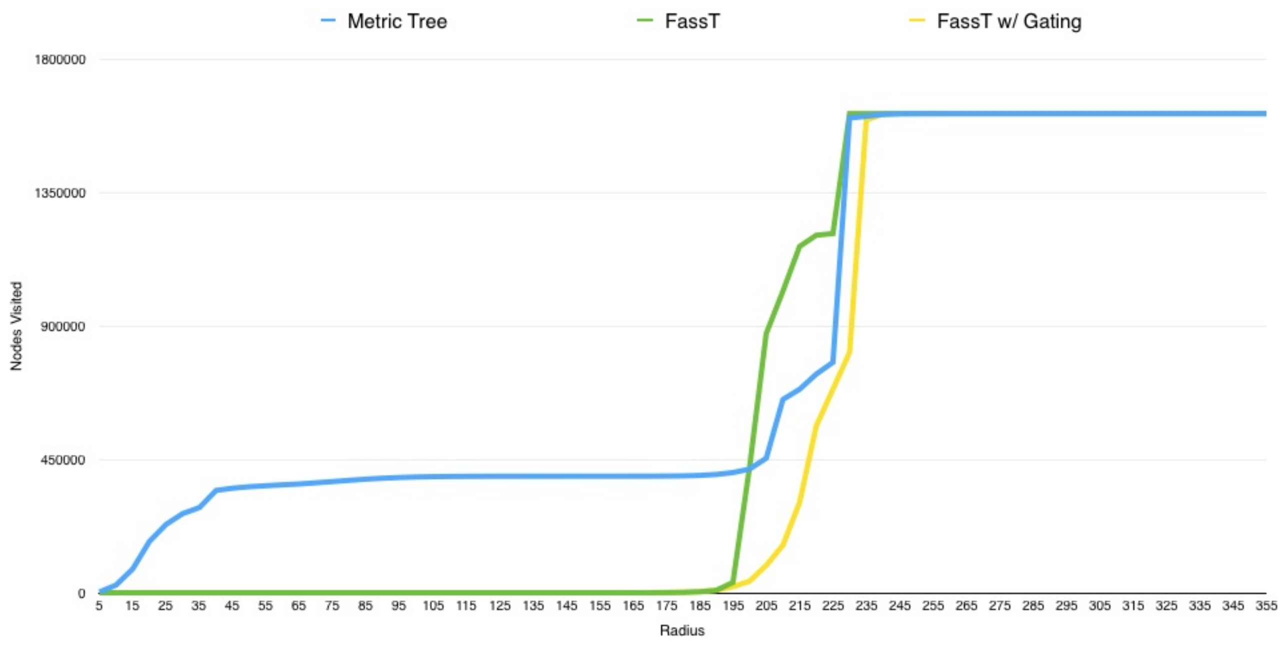

Figure 4.36: Smith-Waterman - Nodes Visited

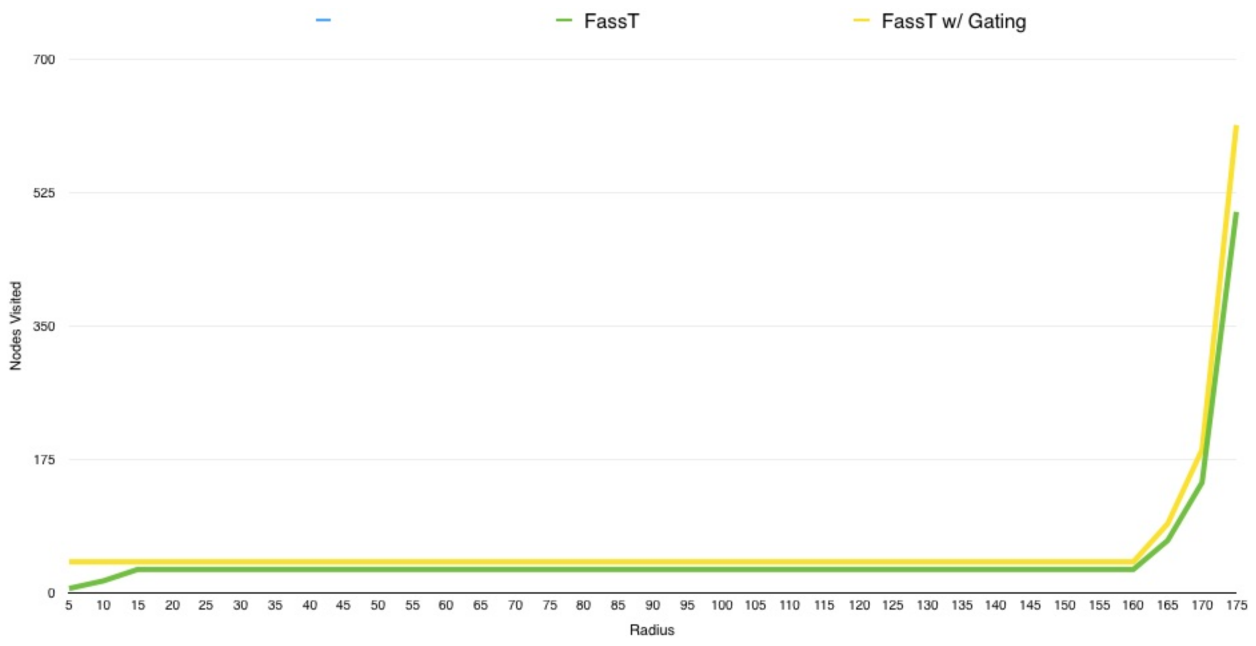

Figure 4.37: Smith-Waterman - Nodes Visited - FaSST and FaSST with Gating

Again, until the limit, looking at the time each tree takes to run a query; the metric tree takes on average 20 seconds to complete a search while FaSST takes on average 4 milliseconds and FaSST with Gating takes 1.2 milliseconds. That is 4737 and 17769 times improvement respectively.

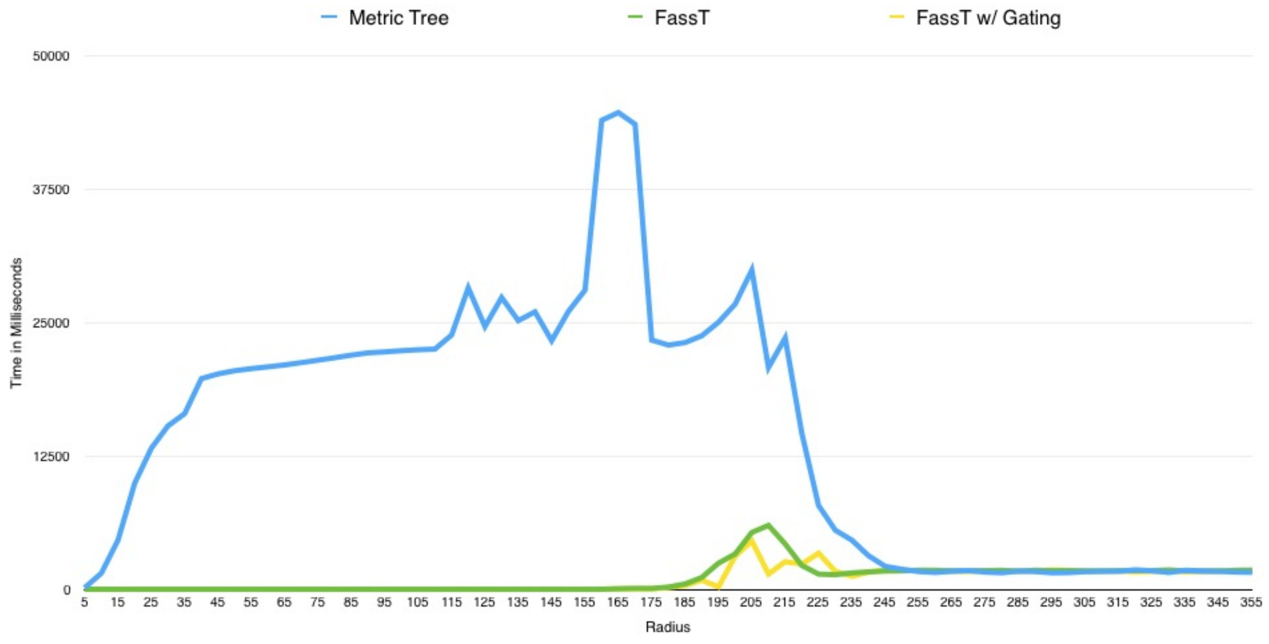

Figure 4.38: Smith-Waterman - Time in Milliseconds

Chapter 5

Summary and Future Work

The Fast Similarity Search Tree presented in this thesis has been shown to be an effective tool for solving range queries in general metric spaces. Its use of bounded search spaces using collections of annular regions proved to be a powerful pruning mechanism. The tests in chapter 4 showed it can provide better search performance than the k-d tree in high dimensional space and beats out the metric tree in all real world scenarios that were tested.

The biggest issue remaining is gating, specifically if and when it should be used. While gating seems like a wanted feature, it appears, in practice, that it may conflict to much with pruning. The result is that many more nodes in the tree are visited than if gating were not to be performed and more distance calculations are made. The issue is that when a node is gated, that gate propagates down the tree to all of its children. Therefore a gate performed near the root of the tree is potentially very harmful where one further down is less so. One idea for future work is to determine if there is a way to strike a balance between gating and performing distance calculations so that the benefits of both may be observed.

Bibliography

[1] Volker Gaede and Oliver Günther. Multidimensional access methods. *ACM Computing Surveys (CSUR)*, 30(2):170–231, 1998.

[2] Christian Böhm, Stefan Berchtold, and Daniel A Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Computing Surveys (CSUR)*, 33(3):322–373, 2001.

[3] Eamonn Keogh and Abdullah Mueen. Curse of dimensionality. In *Encyclopedia of Machine Learning*, pages 257–258. Springer, 2011.

[4] George S Lueker. A data structure for orthogonal range queries. In *Foundations of Computer Science, 1978., 19th Annual Symposium on*, pages 28–34. IEEE, 1978.

[5] Bernard Chazelle. Lower bounds for orthogonal range searching: I. the reporting case. *Journal of the ACM (JACM)*, 37(2):200–212, 1990.

[6] Thomas H Cormen. *Introduction to algorithms*. MIT press, 2009.

[7] Mark De Berg, Marc Van Kreveld, Mark Overmars, and Otfried Cheong Schwarzkopf. *Computational geometry*. Springer, 2000.

[8] Prabhakar Ragde. Simple balanced binary search trees. *arXiv preprint arXiv:1412.4882*, 2014.

[9] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[10] Jon Louis Bentley and Jerome H Friedman. Data structures for range searching. *ACM Computing Surveys (CSUR)*, 11(4):397–409, 1979.

[11] Luc Devroye, Jean Jabbour, and Carlos Zamora-Cura. Squarish k-d trees. *SIAM Journal on Computing*, 30(5):1678–1700, 2000.

[12] Micheál Searcóid. *Metric Spaces*. Springer, 1st edition, 2007.

[13] Edgar Chávez, Gonzalo Navarro, Ricardo Baeza-Yates, and José Luis Marroquín. Searching in metric spaces. *ACM computing surveys (CSUR)*, 33(3):273–321, 2001.

[14] Yang Xu, Zong. *Applied Data Mining*. CRC Press, 1st edition, 2013.

[15] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009.

[16] Meenakshi Sharma and Anjali Batra. Analysis of distance measures in content based image retrieval. *global journal of computer science and technology*, 14(2), 2014.

[17] Christian Beecks, Merih Seran Uysal, and Thomas Seidl. Signature quadratic form distance. In *Proceedings of the ACM International Conference on Image and Video Retrieval*, pages 438–445. ACM, 2010.

[18] Eric Sven Ristad and Peter N. Yianilos. Learning string edit distance. *IEEE Transactions on Pattern Recognition and Machine Intelligence*, 20(5):522–532, May 1998.

[19] Sudipto Guha, HV Jagadish, Nick Koudas, Divesh Srivastava, and Ting Yu. Approximate xml joins. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 287–298. ACM, 2002.

[20] Gregory Cobena, Serge Abiteboul, and Amelie Marian. Detecting changes in xml documents. In *Data Engineering, 2002. Proceedings. 18th International Conference on*, pages 41–52. IEEE, 2002.

[21] V Levenstein. Binary codes capable of correcting spurious insertions and deletions of ones. *Problems of Information Transmission*, 1(1):8–17, 1965.

[22] Dongwon Lee. *Query relaxation for xml model*. PhD thesis, University of California Los Angeles, 2002.

[23] Jeffrey Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, 40:201–213, 1991.

[24] Pavel Zezula, Giuseppe Amato, Vlastislav Dohnal, and Michal Batko. *Similarity search: the metric space approach*, volume 32. Springer Science & Business Media, 2006.

[25] Jeffrey Uhlmann. cascading metric tree notes. unpublished.

[26] Richard W Hamming. Error detecting and error correcting codes. *Bell System technical journal*, 29(2):147–160, 1950.

[27] Temple F Smith and Michael S Waterman. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, 1981.

[28] Eugene G Shpaer, Max Robinson, David Yee, James D Candlin, Robert Mines, and Tim Hunkapiller. Sensitivity and selectivity in protein similarity searches: a comparison of smith–waterman in hardware to blast and fasta. *Genomics*, 38(2):179–191, 1996.

[29] Datasets - dna analytics. http://www.genomics.agilent.com/article.jsp?pageId=2079. Accessed: 2016-02-16.