Empirical evaluation of metric indexing methods

Rune Fevang
Arne Bergene Fossaa

Master of Science in Computer Science
Submission date: June 2008
Supervisor: Magnus Lie Hetland, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

The Master thesis involves implementing prototype versions of existing metric indexing methods, and possibly some new variations of these, and evaluating them experimentally. A part of this evaluation involves obtaining real-world and synthetic data sets with appropriate distance functions.

Assignment given: 15. January 2008
Supervisor: Magnus Lie Hetland, IDI

Abstract

Metric indexing is a branch of search technology that is designed for search non-textual data. Examples of this includes image search (where the search query is an image), document search (finding documents that are roughly equal) to search in high-dimensional Euclidean spaces. Metric indexing is based on the theory of metric spaces, where the only thing known about a set of objects is the distance between them (defined by a metric distance function). A large number of methods have been proposed to solve the metric indexing problem. In this thesis, we have concentrated on new approaches to solving these problems, as well as combining existing methods to create better ones.

The methods studied in this thesis include D-Index, GNAT, EMVP-Forest, HC, SA-Tree, SSS-Tree, M-Tree, PM-Tree, M*-Tree and PM*-Tree. These have all been implemented and tested against each other to find strengths and weaknesses.

This thesis also studies a group of indexing methods called hybrid methods which combines tree-based methods (like SA-Tree, SSS-tree and M-Tree), with pivoting methods (like AESA and LAESA). The thesis also proposes a method to create hybrid trees from existing trees by using features in the programming language.

Hybrid methods have been shown in this thesis to be very promising. While they may have a considerable overhead in construction time,CPU usage and/or memory usage, they show large benefits in reduced number of distance computations.

We also propose a new way of calculating the Minimal Spanning Tree of a graph operating on metric objects, and show that it reduces the number of distance computations needed.

Preface

This thesis is the culmination of five years education at the Institute of Informatics at the Norwegian University of Science and Technology.

This thesis is part of the iAD (Information Access Disruption) - project, which is a collaboration between FAST Search and Transfer and NTNU.

We would like to thank Magnus Lie Hetland for his help and support with the thesis.

Through this project we managed to find an error in the sorting method in the standard library for the Jython programming language. Although this did not directly influence this thesis (it was found by accident while trying to debug a method), we mention it here to show the work done. The bug-report can be found at `http://bugs.jython.org/issue1835099`

Parts of this thesis has used text from the fall project of 2007, which had the same focus as this thesis. These parts are Section 2.1.2 about AESA/LAESA , 2.3.6 about CM-Tree and parts of the Introduction chapter.

Contents

Chapter 1

Introduction

1.1 Goal of the Master thesis

The goal of this project is to investigate and implement new methods and approaches to indexing metric data, as well as benchmarking these methods to see if they show an improvement when compared to existing methods.

1.2 Background

Search technology is the art of trying to find matches to a query in data, and at the same time try to keep searches efficient and accurate. Traditionally, the notion of search technology was understood to only include textual search, where the search query is a word or a phrase, and the data is a collection of documents. Search on non-textual data has usually been done by searching meta data, or by searching the context around the data. An example of this is Google Image Search, which searches the HTML document where the image is located, and does not look at the image itself.

For search in non-textual data where textual meta data is lacking, other approaches has to be taken. One approach that has seen significant promise is metric indexing.

1.3 Metric Indexing

Metric indexing is a method for indexing data where little or nothing is known about the data, except for a distance function between objects. To be used with metric indexing methods, the distance function has to have the following properties:

$$d : \mathbb{X} \times \mathbb{X} \rightarrow \mathbb{R}^{+} \tag{1.1}$$

$$d(x, y) = 0 \iff x = y \tag{1.2}$$

$$d(x, y) = d(y, x) \tag{1.3}$$

$$d(x, z) < d(x, y) + d(y, z) \tag{1.4}$$

This distance function and the objects together create a metric space. The metric space is usually described a the tuple (U, d), where U is the universe of objects and d is the distance function over the objects.

The most important property when using metric indexing is the triangle inequality(1.4). Knowing this, we can exclude objects that we know are outside thre range of our query, without actually doing the distance computation itself (which might be an expensive operation).

This makes metric indexing good for non-domain-specific indexing, with the ability to plug in a type of data and make it indexable without worrying about what type of data is indexed. Of course, this is not always true - some data structures will have a structure that makes it fast with one kind of indexing, but slow with another.

In our master thesis we will try to document different indexing methods, as well as do tests to find which ones shows the best promise for different metrics. A weakness with many other tests of metric indexing methods is that it has usually been done on distances between vectors. A problem with this is that metric indexing often is meant for problems that is more advanced (and more expensive), and where the metric space doesn't necessarily share any resemblance with an euclidian space.

1.4 Space Division

In this section we present different methods to divide up a metric space into partitions. Most of the metric indexing structures we present later use one or more of these methods to organize their data.

1.4.1 Ball Partitioning

Ball partitioning is a basic way of dividing up a metric space. The partitioning is defined by a ball (c, r), where c is the center object of the ball, and r is the radius. All objects where $d(o, c) <= r$ is defined to be inside the ball. All objects where $d(o, c) > r$ is defined to be outside the ball.

A variant of ball partitioning is to partition into multiple shells. This is done by defining k balls (c, r_i)(where $r_i < r_j \iff i < j$). The space is then divided into shells by the limit of each ball, so that all objects where $r_i < d(o,c) <= r_{i+1}$ falls into shell $i+1$. Objects where $d(o,c) <= r_0$ falls into shell 0, while objects where $r_{k-1} < d(o,c)$ falls into shell $k-1$.

Excluded Middle Partitioning

A special case of multiple shell partitioning is the excluded middle partitioning (EMP). With EMP we use two balls $((c, r-\rho), (c, r+\rho))$ to divide the space, where ρ is a constant that defines the width of the excluded middle, and r is the division radius. Using the triangle inequality, we know that if $d(c,o) \leq r$, o has a distance of at least ρ to all objects in the outer shell. Similarly, if $d(c,o) \geq r$, o has a distance of at least ρ to all objects in the inner shell. This property is exploited by several indexing methods.

1.4.2 Hyperplane Partitioning

Hyperplane partitioning works by selecting two objects as pivots, and then partition the objects depending on the distance from each pivot. The division may either divide the space so that that an object belongs to the pivot that is the closest, or it may divide it so that there is an equal amount of objects belonging to each pivot. This is done by first calculating the distance between the pivots and all objects, finding the value $\frac{d(o,p_1)}{d(o,p_2)}$ for all objects o. The median of these values are found, and the objects are split accordingly.

The name hyperplane refers to the boundary between the two sets, which forms a hyperplane if the space is Euclidean.

1.4.3 Voronoi Diagram

A Voronoi diagram of a metric space U is a decomposition defined by objects in a subset $S \subseteq U$. The space is divided into parts called Voronoi Cells, where the Voronoi Cell of $s \in S$ is defined so that every object in the cell is closer to s than to any other object in S. Formally it can be described as:

$$VC(s) = \{p \in U | \forall s' \in S - \{s\}, d(s', p) > d(s, p)\}$$

The Voronoi diagram is a generalization of hyperplane partitioning, since a Voronoi Diagram where $|S| = 2$ is the same as a hyperplane partition.

Some objects may fall on the edge between two or more cells (if $d(t,p) = d(s,p)$), and are not technically in any cell. In a metric indexing implementation of a Voronoi Diagram decomposition, these are usually just assigned to one of the cells.

1.5 Examples of Metric Spaces

In this section we will present a few of metric spaces. This is obviously not intended to be a comprehensive list, but the selected distances are the ones we have focused on in this thesis.

1.5.1 Minkowski Distance

Minkowski distances is a group of distances over an Euclidean space, which includes the standard Euclidean distances. The different Minkowski distances are usually denoted as L_p, where L_p is defined as

$$L_p = \sqrt[p]{\sum_{i=1}^{n} |x_i - y_i|^p}$$

L_1 is more often known as the Manhattan distance, L_2 as the standard Euclidean distance, and L_∞[1] as the Chebyshev distance [2]. The running time of a distance computation is $O(k)$, where k is the number of dimensions in the space.

How the distances are defined in a plane can be seen in Figure 1.1 to Figure 1.4, where the green dot represents the centre, and the red line represents the set of all points which are equidistant to the centre.

1.5.2 Edit Distance

The edit distance is defined as the work that is needed to transform one string into another, using only insert (insert a character somewhere in the string), delete (remove a character from the string) and replace (replace one character with another) operations. In the generalized version it is also possible to apply weights to each of these operations, and give different weights depending on which character is inserted/replaced/deleted.

For the edit distance to be a metric, a few restrictions have to be imposed. First, the cost of a delete must be equal to the cost of an insert(so that symmetry holds). Second, the cost of replacing any given character a with any given character b has to be the same as replacing character b with a.

[1] $L_\infty = \max_{i=1}^{n} |x_i - y_i|$

[2] This distance is also called the chessboard distance, named because this is how one would measure how far a king in the game of chess would have to move to get from one square to another.

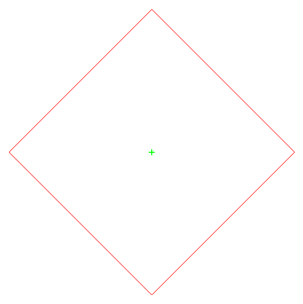


Figure 1.1: L_1 metric

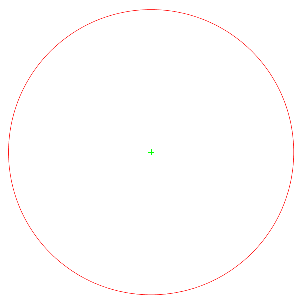


Figure 1.2: L_2 metric

The running time of a distance computation is $O(nm)$ where n is the length of the first string, and m is the length of the second string.

1.5.3 Quadratic Form Distance

The Quadratic Form Distance (QFD) is a distance function that computes the difference between two histograms by comparing every bin in the first histogram with every bin in the second histogram. The correlation between two bins in the histograms (defined by a correlation function) is used to weight the distance between the two bins. QFD may be used with multidimensional histograms.

$$QFD(a,b) = \sum_{0 \le i,j < n} |a_i - b_i|\,|a_j - b_j|\,corr(i,j)$$

QFD is generally not a metric, but according to [NBE$^+$93], if $corr(i,j) = 1 - d(i,j)/d_{max}$, where $d(i,j)$ is the euclidean distance between bin i and bin j (in the histogram), then QFD is a metric.

In this Master thesis, we have implemented QFD to measure the difference between images. Each image is first converted to points in a 5-dimensional space, 3 dimensions (L,a,b) for colors in the CIELAB colorspace, and 2 dimensions (x,y) for the plane coordinates. A 5-dimensional histogram is then created for each image by simply dividing the space into 5-dimensional bins of equal size.

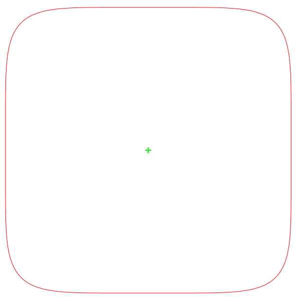


Figure 1.3: L_6 metric

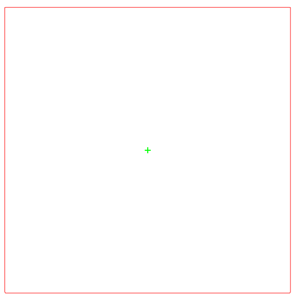


Figure 1.4: $L_{\inf}$ metric

We choose $d(i,j) = \sqrt{l^2 + a^2 + b^2 + \gamma(x^2 + y^2)}$, where γ is a value between 0 and 1 that defines how much the plane coordinate dimensions influences the distance.

Chapter 2

Metric Indexing Methods

2.1 Pivoting Methods

Pivoting methods is a family of indexing methods that at indexing time finds the distance between all objects in the metric space and a set of objects called pivots(this set may be a subset of the objects or a totally disjoint set). The distances are used when searching by first finding the distance between the query object and all pivots, and then pruning out objects that cannot be in the query (usually by using the triangle inequality to find a lower bound for the distance between the query and the objects).

2.1.1 AESA

Approximating and Eliminating Search Algorithm (AESA), first mentioned in [Vid86], is an indexing method where all objects can be looked at as pivot objects. When indexing, the distance for every pair of objects in the data set is computed and saved in a matrix. When doing a range search, a random pivot is chosen and the distance from the query to this pivot is computed. A lower bound distance for all the objects in the data set is computed by using the equation $|d(q,p) - d(p,o)| < d(q,o) \rightarrow d_{lo}(q,o) = |d(q,p) - d(q,o)|$. All objects where $d_{lo}(q,o) < r$ are then removed from the set of possible results. A new pivot is chosen from the the result set, and the lower bound is updated.

When choosing a pivot, the pivot which has the lowest lower bound is usually chosen. The rationale behind this is that the lower bound is used as a heuristic for the actual distance to the object. If the actual distance is low, it is hypothesized that there is a higher chance that it will remove a higher number of objects. This isn't necessarily true, but works for data sets where the distribution of distances is fairly uniform.

When searching with AESA, it is also possible to keep track of the upper bound ($d_u = d(q,p) + d(p,o)$). If $d_u(q,o) <= r$, one can add the object to the result set without actually computing the distance to it.

When compared to other methods, AESA performs up to an order of magnitude faster than other methods, and for many data sets it runs in $O(1)$ distance computations. This comes at a cost, however, memory usage is $O(n^2)$, and the extra CPU time when running a query is up to $O(n^2)$.

The running time can be reduced with usage of Reduced Overhead AESA (ROAESA)[Vil85].

2.1.2 LAESA

The main drawback of AESA (CPU and memory usage) is addressed by Linear AESA (LAESA)[MOC96]. LAESA is based on AESA, but has as the main difference that it uses only a fixed number of pivots when indexing the dataset. This reduces the memory usage to $O(nm)$,where m is the number of pivot values. With $m = n$, LAESA is essentially AESA. With a smaller m we get a higher number of distance computations when performing searches, meaning that LAESA is trade-off between CPU/memory and distance computations. The number of pivots that is needed to get a reasonable running time (compared to AESA) is not necessarily known, and may be different in different depending on the nature of the metric space.

Pivot Selection

The selection of good pivots is important to achieve good performance for indexing methods with a fixed number of pivots (like LAESA) [BNC01]. The original selection method for these methods was to choose the pivots randomly. This may however produce suboptimal pivots. An example is if the pivots that are chosen are close to each other. In a query, the pivots will then prune the same objects, making it meaningless to check both pivots.

To improve this we can use heuristics when choosing pivots. One of the most obvious heuristic is to try to get pivots as far away from each other as possible. This can be done by first choosing a random pivot as the first pivot. We then choose the object that is the furthest from this object as the next pivot. We continue is the same manner, always choosing the object where the minimum of distances to the previous pivots are maximized. Generally, it can be described by the following equation:

$$\max_{o \in D} \left(\min_{p \in P} d(o,p) \right)$$

Another way of improving the pivot selection is the Bustos method[BNC01]. This is basically a method that tries to maximize $|d(q,p) - d(q,o)|$ for objects. The rationale of this is that a high value here will result in a higher lower bound distance for objects, meaning that more objects will be pruned in a query.

The lower bound for the distance between a query and an object is then

$$D_P(q,o) = \max_{p \in P} |d(p,q) - d(p,o)|$$

where P is the set of all pivots.

The Bustos method tries to maximize this iteratively by first starting with an empty set P of pivots. It then samples N different objects from the data set in each iteration and tries to find the object n from these objects that maximize

$$\sum_{(q,o) \in A} D_{P+\{n\}}(q,o)$$

where A is a set of sample queries, picked at random from the data set.

2.2 Tree Based Methods

Tree based methods generally divide up the query (by a given partition method) into different sets, and then recursively divides up each subset in the same way. The methods vary mainly in the following ways:

Division method: The tree can divide in many different ways on each level, using balls, hyperplane division or Voronoi division.

Branching: The number of branches on each level varies from method to method, and the amount may either be static or dynamic (dependent on the metric space).

2.2.1 SA-Tree

Spatial Approximation(SA), first mentioned in [Nav99], is a method that tries to view all objects as nodes in a Voronoi graph[1]. When searching one moves from object to object on the graph so that the next object is always closer to the

[1]There isn't a definite definition on what constitutes a Voronoi graph in a metric space. In Euclidean space, it is the graph between centers in a Voronoi diagram, where the edges of the graph are between neighbours. It isn't possible to transfer this directly to a metric space. This is because there may exist an edge (a hyperplane) that is not part of the set to be indexed (but that is part of the universe, and may thus be a query object), which makes two objects neighbours. Some metric spaces may also not have a Voronoi graph at all. A more thorough discussion of this can be found in the mentioned paper.

query. When an object is reached so that it is not possible to move to another object that is closer to the query, we know that the object is the closest.

This idea is generally not good for a non-Euclidean metric space, since there is not enough information in just distance operations to make a Voronoi graph (according to [Nav99]). This means that when we try to find out if any object is closer to the query than our last object, we may have to compute the distance to all other objects to be sure.

The SA-tree tries to solve this by only using this method on a subset of the objects, where more information is known. For each node in the tree, a center object $a \in S$ and a list of neighbouring objects $N(a)$ is defined. This list of neighbours have the following property:

$$\forall x \in S, x \in N(a) \Leftrightarrow \forall y \in N(a) - x, d(x, y) > d(x, a) \tag{2.1}$$

This means that $N(a)$ is a set of objects so that every neighbour is closer to a than to any of the other neighbours. The tree is defined recursively, so that each neighbour becomes the center of a new tree. The remaining objects are then assigned to the neighbour that is the closest. We also keep track of the maximum distance $R(a)$ to any object in any sub-tree (this is to improve efficiency at query time). The tree-construction continues until there is only one object left in each tree.

It is easy to see that there are several sets of neighbours that will fulfill (2.1), and finding the best set (the one with the fewest neighbours) is not trivial. The method proposed by [Nav99] is to sort all objects by the distance to the center object, and then add objects to the neighbour list if they don't break (2.1).

When we do a range search we first compute the distance between the query object and all objects in the set $a \cup N(a)$ (and add any object that has as distance less than range to the result-set). We then find the object that is the closest, c. We also have a variable t (tolerance) that is initially set to $2 \cdot range$ when the root is queried. We then only go further down in the trees $\{\forall b \in N(a), d(b, q) - t < d(c, q)\}$, and recursively search each tree with $t_{child} = t_{parent} - (d(b, q) - d(c, q))$.

Variations

Several variations of SA-tree has been proposed. The Dynamic SA-tree (DSA-tree)[NR01] is a dynamic version of SA-tree, which has almost the same number of distance operations as SA, while adding the possibility of adding and removing objects from the tree dynamically. The HSA - tree (Hybrid SA-tree)[AMNR03], is a method that combines SA-tree with pivoting, using the center objects as pivots in the tree.

2.2.2 Sparse Spatial Selection Tree

Sparse Spatial Selection (SSS) is a method originally developed to find good pivot objects (see [BFPR06]) in pivot-based methods like LAESA. It works by only adding new pivots if they are a certain length from all other pivot objects so selected far.

The method goes through the list of objects and adds an object to the list of pivots if the distance to all pivots chosen until then is more than $0.4{d_{max}}$[2], where d_{max} is the maximum distance between any two objects in the list of objects (in reality, d_{max} is approximated to $2r^c$, where r^c is the covering radius of a given center object).

SSS is unique in that it automatically decides how many objects should be chosen as pivots, by automatically having more pivots if the intrinsic dimensionality is high.

The SSS-tree[BPS$^+$08] is a tree based on the SSS pivot selection method, by using SSS on each level of the tree to find out which objects should be chosen as centers on the next level of the tree. The tree is then built recursively, by first putting new objects into a bucket, and then creating a tree when the size of the bucket is above a given maximum bucket size. The maximum distance from the center object to an object in a sub-tree is saved when creating the tree.

A range search is performed by recursively going down the nodes where the range ball intersects with the ball of the tree (the ball with the center of the tree as the center object, and the maximum distance as the radius). When a bucket is reached, a standard linear search is done.

Variations

The SSS-tree is quite new (at the time of writing, the paper that mention it has not yet been published), but it has an obvious improvement when it comes to the bucket, which could be replaced with an AESA/LAESA implementation instead of a linear search.

2.2.3 Geometric Near-Access Tree

Geometric Near-Access Tree (GNAT) was first proposed by [Bri95]. The method works by recursively dividing up the metric space into a Voronoi diagram This is done by first selecting k objects as pivots (called split points in [Bri95]). The selection of pivots may be random, or a general pivot selection technique as explained in 2.1.2 may be used. The pivots are then indexed in a similar way to AESA, computing the distance between every pair of pivots. The remaining

[2]Why 0.4 is chosen is explained in [BFPR06]

objects (the objects not selected as pivots) are assigned to the pivot which is the closest, and then indexed recursively in the same way.

Search is done by going through all pivots to see if the query ball intersects with the Voronoi Cells of the pivots. An AESA-like method is used to keep the number of distance operations at a minimum, so that cells completely outside the query ball (which is determined using triangle inequality) is not considered.

Since the tree may become very unbalanced, the authors proposes to make the number of pivot objects further down in the tree dynamic. This is done by first assigning a degree of k to the root node. Then each sub-tree get a degree k' depending on the amount of objects applied to that sub-tree such that the average degree is k. This ensures the balance of the tree.

Variants of GNAT includes EGNAT, which saves the distance from the center object to all objects (which may speed up distance evaluations in the leaves). We also propose a further generalization of EGNAT called AGNAT, which transforms into an AESA structure if the size of the data set gets below a certain value.

2.2.4 Excluded Middle Vantage Point Forest

The Excluded Middle Vantage Point Forest (EMVPF) structure is a tree based method that optimizes search for certain range, defined when indexing. It is based on the Vantage Point (VP) tree, which works by recursively dividing up the space in two by using ball partitioning. Usually the median distance is used to divide the two spaces.

The Excluded Middle VP (EMVP) -tree is generalisation of this. Instead of using ball partitioning, it uses Excluded Middle Partitioning. A value ρ is chosen, and on each level all objects where $med - \rho < d(o, c) < med + \rho$ are removed. The objects which are removed are put in a new tree, which is then recursively indexed, making it a forest of trees.

Search is done by going through each tree in the forest separatly. Then on each node, the distance to the center object is computed. If the distance is less than the query range, we add the center object to the result set. We then recurse the left sub-tree if $d(q, c) <= med - \rho + r$, and the right sub-tree if $d(q, c) >= med + \rho - r$ (where q is the query and r is the range). If $r < \rho$ we can guarantee that only one sub-tree will be traversed.

A more extensive analysis can be found in [Yia98].

2.2.5 Hybrid methods

Tree-based methods are capable of excluding large parts of the search space at once (for example, pruning out whole subtrees when it is known that the objects cannot be in the search range), but when the search gets closer to the leaves the search is often less effective, often requiring to query all objects of a sub-tree.

Hybrid methods try to combine the good aspects of tree-based methods (small memory footprint, less IO, etc.) with pivot-based methods (small number of distance computations).

One way of implementing this is to go from a tree-based to pivot-based implementation when the number of objects in a sub-tree get below a certain threshold. The objects of this sub-tree (usually called a bucket) is then indexed using a pivot-based method like AESA or LAESA.

A variant of this hybrid method is to use the fact that when an object in the tree is reached, it is usually known for sure that the distance between the query object and a certain set of objects (usually the ancestors of that object) has already been computed. If these objects are known at indexing time they can be used as pivots for the object. Pivot filtering can then be added with no cost (in terms of distance computations) since the distance to the pivot objects has already been computed.

2.3 M-Tree Based Methods

Methods in the M-Tree family of methods are, as the family name implies, tree-based methods. However, because of the large number of methods that have been proposed within this family, they are discussed in their own section.

M-Tree based methods are designed to be stored on disk, and to not only reduce CPU-costs (by lowering the number of distance computations), but also reduce I/O-costs (by reducing the number of disk pages that need to be accessed). They attempt to accomplish this by storing each node in the tree in a single disk page. This puts a natural limitation on the size of each node.

M-Tree based methods generally support both dynamic insertion and deletion of objects, without incurring the large reconstruction costs that often plague other methods [ZADB05]. M-Trees are grown in a bottom up fashion, in the same way that R-Trees [Gut84] and B-Trees [Com79] are. Objects are inserted into leaf nodes, and the leaf nodes are then split if they get too large. If the parent node get too large after a split, it in turn is split, and this propagates up the tree if necessary. If the root node is split, a new root node is allocated, and the tree grows up one level. This way of growing the tree ensures that it remains balanced.

2.3.1 M-Tree

The standard M-Tree was first proposed in [CPZ97]. The M-Tree design is pretty simplistic, and has partly because of this proved to be an excellent basis for other ideas to expand on.

Each internal node in the tree, has an associated pivot object and a covering radius. These two properties acts as a kind of definition of the node. Every object contained in the node or its sub trees is guaranteed to have a distance of no more than the covering radius from the pivot. When performing searches on the tree, these properties of a node can be used for pruning.

Internal nodes in the M-tree store up to m tuples $\langle p, r^c, d(p, p^p), ptr\rangle$ representing child nodes. p is the child's pivot, and r^c is the corresponding covering radius. p^p is the parent's pivot, and ptr is a pointer to the child node. The distance $d(p, p^p)$ is stored to enhance pruning when searching the tree.

Leaf nodes' entries do not point to child nodes. r^c and ptr are therefore omitted, and entries are stored as tuples $\langle p, d(p, p^p)\rangle$.

When a new object o is to be inserted into an M-Tree, a suitable leaf node has to be located. The leaf node should ideally be one that contains other objects that are close to o. The search for the leaf node starts at the root, and descends down the tree until a leaf node has been selected. At each level, the child node that requires the least extension of its covering radius to accommodate the new object is selected. If there is more than one node that require no extension of its covering radius, the node that minimizes the distance $d(p, o)$ is selected.

There are several possible strategies proposed that the M-Tree can use when splitting a node. The strategy the authors outlined as the one yielding the best tree is `mM_RAD_2`. This strategy selects the split that results in the maximum covering radius of the new nodes being the lowest.

Range Searching with query object q and radius r is done as follows: The search starts at the root node. Each entry $\langle p, r^c, d(p, p^p), ptr\rangle$ is examined. If $|d(q, p^p) - d(p, p^p)| - r^c > r$, we don't have to calculate the distance $d(q, p)$ in order to know that no objects from the sub tree are in the result set, and we do not have to search the sub tree pointed to by ptr. Otherwise we compute $d(q, p)$. If $d(q, p) - r^c > r$, again we do not need to examine the sub tree ptr points to. If none of these criteria hold, we examine the sub tree recursively.

2.3.2 Slim Tree

The Slim Tree was first proposed in [CTTSF00], and is a modification of the M-Tree. The Slim Tree makes no modifications to the structure of the M-Tree, but changes the construction procedure with three goals in mind. One is to make the tree more compact (reduce the number of nodes in the tree), another is to reduce construction time, and a third is to make the resulting tree more suited for efficient range queries. It tries to accomplish all of this by making three adjustments to the M-Tree construction process.

When an object is to be inserted into the Slim Tree, a suitable leaf node is located in a different way than the one the M-Tree uses. Like the M-Tree, it first tries to find a node that already covers the area where the object is located. But if there is more than one candidate, the node with the minimal occupancy is chosen. The M-Tree on the other hand, would pick the one where the distance to the pivot is minimal. When no nodes cover the incoming object, the node with the pivot that is closest to the object is chosen. This is another difference from the M-Tree, which in this case would have picked the node that required the least radius extension. This new way of choosing suitable leaf nodes tend to fill insufficiently occupied nodes first, thereby helping to make the tree more compact.

Another improvement the Slim Tree offer, is to reduce the relatively heavy construction cost of the M-Tree. Using the `mM_RAD_2` splitting policy, the M-Tree has an $O(n^3)$ time complexity for its splitting operations, where n is the number of entries in a node. The Slim Tree proposes to use a Minimum Spanning Tree (MST) algorithm for finding a good split faster. The idea is to first find the MST of the entries in the node to be split, then remove the longest edge. The two disconnected subtrees remaining will then contain the entries that go into each new node. This approach can sometimes result in an unbalanced split, so the authors suggest picking the most suitable among the longest edges (if there are more of them), where most suitable means the one resulting in the more balanced split. If no good edges are found, we accept the unbalanced split, and delete the longest edge.

The MST splitting algorithm as outlined by the authors, have an $O(n^2 \log n)$ time complexity, and uses $n(n-1)/2$ distance computations. However, as we will see in section 2.3.3, there may be better alternatives.

The final improvement the Slim Tree brings to the table, is the Slim-Down algorithm. The Slim-Down algorithm is applied as a final post-processing step, and aims to reduce the fat-factor of the tree.

Definition 1. Let T be a metric tree with height H and M nodes, $M \geq 1$. Let N be the number of objects. Then the *fat-factor* of a metric tree T is

$$fat(T) = \frac{I_c - H * N}{N} \frac{1}{M - H}$$

where I_c denotes the total number of node accesses required to answer a point query for each of the N nodes in the metric tree.

As can be seen from the definition, the fat-factor is a number between 0 and 1 inclusive, that tells you how much overlap there is between nodes at the same level of the tree. The Slim-Down algorithm tries to reduce the fat-factor in the following way: For each node N in the tree, locate the furthest object/sub node S from the pivot. Then search the siblings of N, to see if any of them completely cover S. If one does, move S from N to the sibling. If S was the only furthest object from N, this will reduce the covering radius of N, without increasing the covering radius of the node S was transferred to. Since S is no longer within the covering radius of N, we have reduced the fat-factor of the tree.

2.3.3 Efficient Metric MST Calculation

The Slim Tree uses a metric Minimal Spanning Tree to provide an efficient way of splitting nodes. However, we noticed during the course of our work that the approach to building the MST used, is inefficient both in algorithm complexity, and in the number of distance computations needed.

In the original Slim Tree paper [CTTSF00], the MST algorithm is presented as having complexity $O(n^2 \log n))$, where n is the number of nodes. However, when using Prim's algorithm with an adjacency matrix implementation, this complexity reduces to $O(n^2)$ [CLRS01]. The standard way of computing the MST of a graph using Prim's algorithm involves computing all $n(n-1)/2$ pairwise distances in the graph. This approach does not however take advantage of the properties of metric spaces to reduce the number of distance calculations.

To take more advantage of these properties, we exploit the fact that not all distances are needed to start running the algorithm. Prim's algorithm works by maintaining a list, storing for each unmarked node, the shortest edge from a marked node to that node. At each iteration of the algorithm, we mark the node N that has the shortest edge leading to it, and include that edge in the MST. Then we update our list of shortest edges, by examining all edges leading from N to unmarked nodes. In this final step, we can avoid computing the distance from N to the unmarked node N_u if there is a node N_p that has already had its distances $d(N_p, N)$ and $d(N_p, N_u)$ calculated, and $|d(N_p, N) - d(N_p, N_u)| \geq D_{short}$ where D_{short} is the shortest edge found leading from a marked node to N_u so far.

Unfortunately, this change of Prim's algorithm changes the complexity from $O(n^2)$ to $O(n^3)$. A possible compromise is to only consider the closest node to N_u found so far as a candidate for N_p. Intuitively, closer nodes are better suited for estimating the distance to the node. Additionally, we are guaranteed to have already calculated $d(N_p, N_u)$ if this node is used. Implementing this approach maintains the $O(n^2)$ time complexity of Prim's algorithm, while it potentially reduces the number of distance computations needed.

2.3.4 PM-Tree

The PM-Tree (Pivoting M-Tree) combines the M-Tree with principles from the LAESA approach (see section 2.1.2). It was first proposed in [Sko04].

In a normal M-Tree, the pivots used are local to each node, and every node entry has only one pivot for possible filtering. The PM-Tree adds a number n_p of global pivots to the structure. n_{hr} ($n_{hr} \leq n_p$) of these pivots are used for entries of internal nodes, and n_{pd} ($n_{pd} \leq n_p$)are used in leaf node entries. Leaf node entries $\langle p, d(p, p^p), DP\rangle$ now also contain the distance from the object to all n_{pd} leaf pivots, represented by the array PD. Similarly, internal nodes get augmented by an array HR of length n_{hr}, and now looks like $\langle p, r^c, d(p, p^p), ptr, HR\rangle$. The HR array however, is not an array of distances. Instead it is an array of pairs of distances, representing the minimum and maximum distance from each pivot to any object contained in the sub tree pointed to by ptr.

When an object is to be inserted into a PM-Tree, the distances to all n_p pivots are computed and remembered first, so that they can be used to update the HR and PD arrays of the tree without computing the same distance more than once. After the node has been inserted though, only the n_{pd} distances to the leaf entry pivots are remembered (in the leaf nodes). This could cause splitting of leaf nodes to become a relatively expensive operation when $n_{pd} < n_{hr}$, since all the distances from objects in the leaf node to pivots that are exclusive to internal nodes, need to be recomputed.

When initiating a range search on the query object q with range r, we can utilize the HR and PD arrays to exclude sub trees, without computing any additional distances (except from the distances from the query object to each of the n_p pivots, computed once before the search starts). When looking at internal node entries $\langle p, r^c, d(p, p^p), ptr, HR\rangle$, we no longer need to examine the sub tree pointed to by ptr if the following expression is false:

$$\bigwedge_{l=1}^{n_{hr}} d(q, p_l) - r \leq HR[l].max \wedge d(q, p_l) + r \geq HR[l].min).$$

Similarly, objects stored in leaf nodes need not be examined if the following necessary condition is not met:

$$\bigwedge_{l=1}^{n_{pd}} (|d(q, p_l) - PD[l]| \leq r).$$

Since the new ways of excluding sub trees don't incur any additional distance computations, they provide a nice advantage over the M-Tree when performing searches.

According to the authors, low values of n_{pd} needs the least number of disk accesses when searching the tree. However, the number of distance computations increases with lower values of n_{pd}. A good compromise is to use $n_{pd} = n_{hr}/4$. The number of disk accesses is in this case about the same as with the M-Tree, while the number of distance computations is up to 10 times as low. Higher values of n_{pd} does not reduce the number of distance computations significantly.

2.3.5 M*-Tree and PM*-Tree

The M*-Tree and PM*-Tree are the result of adding a structure called "Nearest-Neighbor Graphs" (NN-graphs) to the M-Tree and PM-Tree respectively. Nearest-Neighbor graphs and the necessary modifications to the tree structure are described in [SH07]. The modifications needed to make an M-Tree an M*-Tree, and those needed to make a PM-Tree a PM*-Tree are virtually the same. Therefore, we will only describe how to obtain an M*-Tree structure from an M-Tree one.

The NN-graph structure is relatively simple. Every entry in a node contains information about which other entry in the same node is the closest one, denoted by $NN(p)$, and how far away it is, denoted by $d(p, NN(p))$. Internal node entries now are of the form $\langle p, r^c, d(p, p^p), ptr, \langle NN(p), d(p, NN(p)) \rangle \rangle$, and leaf node entries $\langle p, d(p, p^p), \langle NN(p), d(p, NN(p)) \rangle \rangle$. The purpose of the NN-graph is to provide extra pivots for search pruning. As pivots closer to elements examined are better for pruning than pivots farther away, NN-graphs provide each entry with a potential pivot that is very good.

When searching a node in a M*-Tree, we gain an additional way of pruning the search. When looking at an internal node entry, we can avoid searching the sub tree pointed to by ptr if we have already computed the distance from the query object q to $NN(p)$, and $|d(NN(p), q) - d(p, NN(p))| > r^c + r^q$. Since this new way of pruning cannot be done if we haven't already computed the distance $d(NN(p), q)$, we have to "sacrifice" some entries in order to potentially prune others.

Some entries may be the nearest neighbors of several other entries, while other entries may not be the nearest neighbors of any entries. Clearly, the first type of entry will be much better suited for a sacrifice than the latter, since it allows many more opportunities for other entries to be "saved" later. The authors therefore propose using a "sacrifice queue" to determine the order of calculating distances to entries. The entries first in the queue are those that the most other entries have as their nearest neighbor. When an entry is extracted from the sacrifice queue, all entries that have that entry as their nearest neighbor check if they can be pruned. The entries that can't be pruned are moved to the front of the sacrifice queue. Since it already has been determined that they can't be pruned, they are now perfect candidates for sacrificing.

2.3.6 CM-Tree

The Clustered Metric Tree(CM-Tree)[3], as presented by [AS07] is another variation of the M-Tree structure. The CM-Tree offers several improvements over previous M-Tree structures, the most important of which are outlined below.

Pairwise distance table

The biggest difference between the CM-Tree and other M-Tree structures is the presence of a pairwise distance table in the nodes. The pairwise distance table stores the distances between all entries within a node, and is used to significantly reduce the number of distance calculations needed for most of the CM-Tree's operations.

Centered Navigation Object

A CM-Tree node maintains the property that its navigation object (pivot in other M-Tree structures) is always the "central object" in the node. Every operation on the CM-tree that potentially changes a node, also updates its navigation object. This property of the CM-tree helps reduce the covering radius and empty index volumes of nodes.

Object insertion

To find a leaf for insertion of a new object, the CM-tree recurses down from the root. At each step it selects the node that minimizes, in order of priority, the radius extension required to accommodate the new object, proximity of it's navigation object to the object, and the associated covering radius. The pairwise distance table is used actively to reduce the number of distance measures required to find the best path to recurse down.

[3]We didn't implement the CM-Tree in our thesis work. However, we found the CM-Tree to be the best of the methods we implemented for our project last semester. We found it interesting to analyze how the CM-Tree compares to the methods we implemented in this thesis, and have therefore included the description of the CM-Tree from the project report [FF07]

Node split

The CM-tree's split operation differs from that of other M-tree structures in several aspects. It may be triggered from a new "clustering criteria", in addition to when nodes overflow. This is done so that objects that naturally belong in different clusters should also appear in different nodes of the tree.

When splitting a node, not only two, but up to L nodes may be the result of the operation. This is in line with the clustering criteria for splits; when a split is initiated, objects that naturally appear close to each other should go into the same nodes. Objects that aren't close to each other should go into different nodes.

Additionally, the CM-tree offers a new way of splitting nodes. It uses an approach similar to that of the bu-Tree. It starts by assigning each object of the node a separate "cluster". It then iteratively merges the two closest clusters to each other, until only two clusters remain. At each step, if the number of clusters are less than or equal to L, the clustering is remembered as a candidate partition.

When all the merges are done, the best partition is used in the split. Which partition to use is dependent on a weight parameter $W \in [0, 1]$. A high value of W means that having good clusters are most important. A low value means having a compact tree is most important. The correct value of W depends on the application.

2.4 Clustered Methods

Clustered Methods is a familiy of methods that all derive from the List of Clusters(LC)-method. This method divides the objects into a list of clusters, which are basically balls with a certain number of objects in them. If the query intersects with a ball, the distance is computed to all objects in the cluster.

2.4.1 List of Clusters

LC, first mentioned by [CN05], is a simple method that is based on the VP tree. The difference from the VP tree is that it is extremely unbalanced, since every left sub-tree has a fixed size and every right sub-tree is a recursive LC.

LC works by choosing an object c as a center object (usually a random one), and then assigning the h nearest objects in the data set to a cluster represented by c. The rest of the objects in the data set is then divided in the same way, at each step choosing a center object and selecting the h nearest object as objects in the the cluster. The cluster ball is defined as the ball (c, r^c), where r^c is the maximum distance between a center object and an object in the cluster.

When searching, one simply measures the distance to the first center object. If the query ball intersects with the cluster ball, one does a linear search through all objects in the cluster. If the query ball is inside the cluster ball($d(query, c) + range < r^c$), we can discard all the following clusters (since no object that comes later is of interest).

A simple improvement on distance operations is to save all the distance between the center object and all objects in the cluster. When the distance to the center object is known, it is possible to refrain from computing the distance to some of the objects in the cluster.

2.4.2 Hierarchy of Clusters

The Hierarchy of Clusters (HC) method is a generalization of the LC method, first mentioned by [Fre07]. With HC, a binary tree is made by using the LC method on each level, but instead of fixed number h in each cluster, a function $h(n)$ is used to determine how the data set is divided. Each cluster is then divided recursively in the same way, until a minimum size has been reached. Then the objects are stored in the same manner as a cluster is stored in LC.

Search is done in the same way as in LC, but instead of doing a linear search on the objects in a cluster, a recursive search is done on the HC structure in each cluster. When a leaf node is reached, a linear search is performed through all objects in the leaf.

The HC structure resembles a VP(Vantage Point) - tree, if $h(n) = \frac{n}{2}$. But often, an unbalanced tree may be more efficient (this is the rationale behind the LC structure). [Fre07] proposes two functions for $h(n)$, $h(n) = \frac{n}{k}$ and $h(n) = \frac{n^k}{2}$. This structure is also called Unbalanced Hierarchy of Clusters (UHC).

[Fre07] also mentions a structure called Parallel Hierarchy of Clusters (PHC)[4]. This structure works in the same way as HC, but divides up each cluster into k nodes of equal size. The nodes are selected by making a List of Clusters where the bucket size is $h(n)/k$. The tree can be made unbalanced in the same way as HC, thus creating the Unbalanced Parallel Hierarchy of Clusters (UPHC).

HC and its variations are easily combined with AESA, where AESA is used in nodes when the size of the data gets below a certain limit.

2.5 Similarity Hashing

Similarity hashing (SH) was first proposed by [GSZ01] as an alternative to tree-based hashing. SH works by dividing up the space into bins depending on the

[4]This is not a truly parallel method, it is called parallel because it helps with queries that exploit bitparallelism (like certain implementations of edit distance)

result of a hashing function (formally called a ρ-split - function). It is not an indexing method per se, but an model for an indexing method that other methods may be built on. The only presented method that is based on SH is D-Index (which were defined by the same authors that defined SH).

2.5.1 ρ-split

The ρ-split function$(s^{m,p}) \rightarrow \{0..m\}$ is a function on a metric space that divides the metric space into $m+1$ buckets, where any given element in bucket $i < m$ is separated from all elements in bucket $j < m, i \neq j$ by at least 2ρ. All points that cannot fulfill these requirements (they are closer than 2ρ to a point in bucket $i < m$, without being in bucket i) falls in bucket m, a special bucket called the exclusion bucket. This is formalized in [GSZ01] as:

$$s^{m,\rho}(x) < m \wedge s^{m,\rho}(y) < m \wedge (s^{m,\rho}(x) \neq s^{m,\rho}(y)) \Rightarrow d(x,y) > 2\rho \qquad \text{Separable property}$$
$$\rho_2 \geq \rho_1 \wedge s^{m,\rho_2}(x) < m \wedge s^{m,\rho_1}(y) = m \Rightarrow d(x,y) > \rho_2 - \rho_1 \qquad \text{Symmetry property}$$

From these definitions we see that $s^{m,0}(x) \neq m$. This enables us to be sure that for all range queries with $r <= \rho$

$$\forall x \in U, s^{m,0}(q) \neq s^{m,\rho}(x) \Rightarrow d(q,x) > \rho \vee s^{m,\rho}(x) = m$$

The value m is often referred to as the order of the ρ-split function.

Combining ρ-splits

Usually ρ-split functions are binary (the order of the function is 2). But it may also be useful to have functions that have a higher order. This can be obtained by combining different ρ-split functions. This is done by having the intersection of all combinations of non-exclusion-buckets as buckets (in the combined function) and the union of all the exclusion buckets as the exclusion bucket. E.g., if you have two functions$(s_1^{2,\rho}, s_2^{2,\rho})$, the combined function would have four non-exclusion buckets: $\{s_1^{2,\rho} = 0 \wedge s_2^{2,\rho} = 0, s_1^{2,\rho} = 0 \wedge s_2^{2,\rho} = 1, s_1^{2,\rho} = 1 \wedge s_2^{2,\rho} = 0, s_1^{2,\rho} = 1 \wedge s_2^{2,\rho} = 1\}$ while the exclusion bucket would consist of $\{s_1^{2,\rho} = 2 \vee s_2^{2,\rho} = 2\}$. The order of this combined function is 4 (since there are $4+1$ buckets).

2.5.2 D-Index

The D-Index was first mentioned in [DGSZ03], and is an implementation of Similarity Hashing which uses Excluded Middle Partitioning (EMP) as its ρ-split function.

Excluded Middle Partitioning is a ρ-split function of order two, and defined in SH-terms as:

$$emp_i^{\rho}(o) = \begin{Bmatrix} d(p_i, o) < d_m - \rho & : & 0 \\ d(p_i, 0) > d_m + \rho & : & 1 \\ d_m - \rho <= d(p_i, 0) <= d_m + \rho & : & 2 \end{Bmatrix}$$

In D-Index, first a certain number of pivot objects are chosen. Then, all distances between the objects in the data set and the pivots are computed. For each pivot the median distance is computed. The median distance is then selected as d_m for the EMP-function for that certain pivot object[5]. The EMP-functions are then combined (as explained in 2.5.1), resulting in a ρ-split function of order 2^{n_p}, where n_p is the number of pivots).

The objects are then put in buckets depending on the result from the combined ρ-split function.

The exclusion bucket is then recursively indexed with D-Index (possibly with a reduced number of pivot points) until a certain number of levels.

When a search is performed, we first find in which bucket the search object is (by computing the ρ-split with $\rho = 0$). We then know that the result object must either be in this bucket or in the exclusion bucket.

Search where r $> \rho$

Sometimes it may be required to do a search even though the search radius r larger than ρ. This problem also occurs if we do a kNN-search where the kth object is larger than ρ from the search object.

To solve this the search function is extended to also work for $r > \rho$. This is done by first finding all buckets that intersect with the query ball. This is done by computing the $emp_i^{r-\rho}(q)$ for the query object q. If any of the ρ-split functions return 2, we have to check for the bucket inside and the bucket outside. In the worst case we will have to go through all buckets (e.g. if $r = \infty$, all buckets has to be accessed).

The primary advantage of D-Index is that as long as the $r < \rho$,the amount of distance operations has a maximum limit, defined by the number of levels and the size of the biggest bucket. If the buckets does not exceed 1 IO-block, then it can be guaranteed that the number of IO-accesses is equal to the number of levels[6]. A disadvantage is that a good choice of ρ will vary from data set to

[5]During testing, it was found that for certain data sets (more specifically edit distance between fairly short strings), a value different than the median can be a good choice for d_m. This may happen if most of the objects in the data set falls in the excluded bucket of the EMP

[6]Assumes that all pivots are in memory

data set, so the data set has to be analyzed before indexing.

Chapter 3

Implementation

3.1 Python

Python is an interpreted programming language, developed by Guido van Rossum. It is a simple, yet powerful language, and is a good tool if you want to develop a solution fast. It does not have the same execution speed as a low level language, but makes up for it in development speed.

3.1.1 Jython

Jython is an implementation of the Python language, written for the Java Virtual Machine. Since Jython compiles all classes into Java classes, it integrates seamlessly with data structures in JAVA (like List and Map), and makes it easy to include Java classes. All one needs to do is to import them as if they were normal Python modules.

In this Master thesis we have exploited this property to write the CPU-intensive parts of indexing function and distance evaluations in the JAVA Language (which is several order of magnitudes faster than Jython), and write the less CPU-intensive (and often more IO-intensive) parts in Jython (which is generally easier to do and means less coding). Regular Python also has this property (modules can be implemented in C), but since we had more experience with the Java language, we chose to use Jython/JAVA.

3.2 Bonsai

One of the goals of this Master thesis was to expand a metric indexing method library called Bonsai. The Bonsai library was started by Magnus Lie Hetland, and tries to implement the whole range of known indexing methods.

The library has a common interface for all methods.

All methods must have a constructor, which accepts a list of objects, as well as a distance function over the objects. It is then assumed that the method will do the necessary indexing of the objects. The method may have several arguments, but all these must have default values, so the method can be instantiated without knowledge of other arguments in the constructor, or the inner workings of the method.

All methods must also have a method `range_search(query,range)`, which does a range search with the given query and range on the index. The values returned can be in any order, and it is not required that distance computations is performed on every object returned (e.g. if the method can guarantee that an object is below the range, without doing the distance computation).

The methods may also have a method `knn_search(query,n)` which should return the n closest objects to the query.

When the preproject to this thesis was started, only a handful of methods was implemented, and the library lacked testing and benchmarking capabilities.

We have extended the library with several methods, all described in the previous chapter. The library has also been extended with support for Edit distance and Quadratic From Distance (it came originally only with Euclidean metric). Since testing and benchmarking support was essential for this project, this was added as well.

3.3 Metrics

The implementation of the metric spaces were done so that they could be used with different methods without changing how the methods work. This means that indexing methods did not have to know the inner working of the distance function to use it.

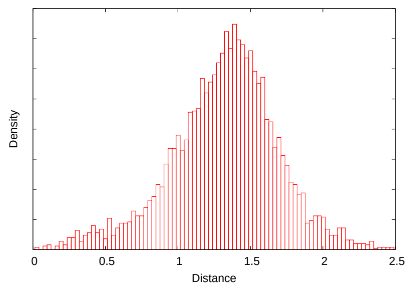


Figure 3.1: Distribution plot for the Clustered Euclidean space

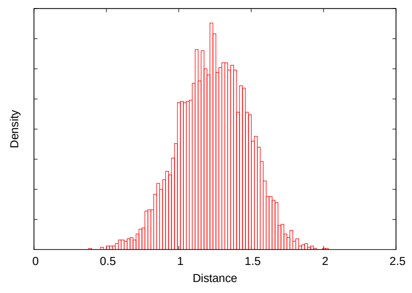


Figure 3.2: Distribution plot for the Euclidean space

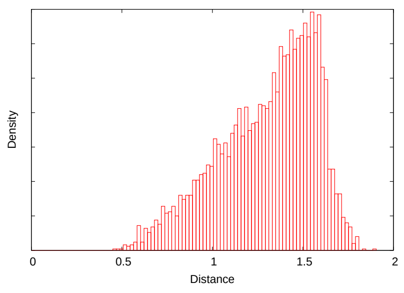


Figure 3.3: Distribution plot for the QFD space

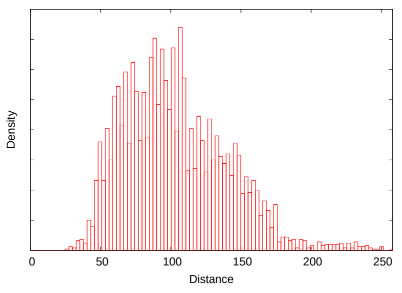


Figure 3.4: Distribution plot for the Edit Distance space

3.3.1 Euclidean

We implemented the standard random Euclidean space to use in our test. The Euclidean metric is simply vectors of n dimensions where each dimension is in the range $[0, 1]$. The distance used is the L_2 distance from the Minkowski family of distances (see Section 1.5.1).

Generally, the higher the dimensions of an Euclidean space, the harder it is to index. This is because higher dimensionality results in a smaller variation in distances (meaning a higher and more narrow distribution plot). This means indexing methods have a harder task the more dimensions there are. A more thorough discussion of this can be found in [CNBYM01].

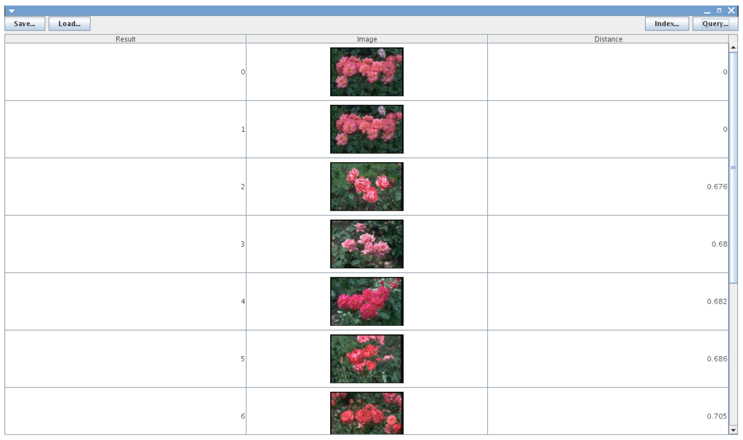
3.3.2 Clustered Euclidean

In real-world spaces, objects often have similar features/are clustered. For example, in an image database (with a good distance function), a set of similar images would be clustered (have a short distance to each other as compared to other objects). To simulate this, we created a clustered euclidean space.

The clustered space is created by first randomly creating n center objects. The remaining objects selects a random center object, and are placed with Gaussian distribution from the cluster center. An object also has a small chance of being placed totally random.

The distribution plot can be found in Figure 3.1. We see that an object has a few objects that are quite near, while the rest of the objects are distributed in almost the same way as the standard random Euclidean distance.

The two Euclidean spaces have a fairly low CPU-cost when it comes to finding the distance between two objects, which means that we are able to test these methods on larger spaces.



Result	Image	Distance
0		0
1		0
2		0.676
3		0.68
4		0.682
5		0.686
6		0.705

Figure 3.5: Screenshot of GUI for QFD. The first object is the query image, and the rest is the results. The query image was part of the index, which is why the first object returned is the same as the query.

3.3.3 Edit Distance

For the Edit Distance metric we have used the Holy Bible, Douay-Rheims Version from Project Gutenberg[bib98] to use as a source of text. The text was divided up in verses, spacing was removed, and the verses was converted to lowercase. The distance function is the same as mentioned in 1.5.2.

3.3.4 Quadratic Form Distance

To create the QFD, we used images from the CLIC [MHGM05] collection of images. The images are divided into groups depending on the motif of the image. The division of the image into multidimensional histograms is configurable. The sizes we used were 4 boxes in L dimension of the color space, 8 for the a and b dimensions, and 2 for the x and y dimensions of the plane. We used 0.5 as the weight for the position.

GUI for Image Search

To aid with developing QFD, we implemented a GUI that could index and query images. The GUI is simple (but enough for our purposes), and shows the 10 nearest neighbours to an image, as well as the distance from the query to each of the resulting images. It is also useful as a stand-alone program, to be used as a demo on how metric indexing can be used with real world indexes.

The program uses the LAESA structure to index images.

3.4 Methods

Here we describe how the different methods have been implemented for Bonsai. Some methods have been omitted, in cases where the implementiation did not deviate from the description in the previous chapter.

3.4.1 D-Index

The D-Index has several implementation tweaks.

First off, the Bustos pivot selection technique from 2.1.2 is used to choose pivots on each level. This is also the method chosen in [DGSZ03] to choose pivots.

Another tweak is how the division radius of the Excluded Middle Partition is chosen. In [DGSZ03], the median is used. This distance may however not be the best division radius. A scenario where this could be the case is if the metric space is heavily clustered. The middle partition may then cover many objects, even though another value for the partition may provide a better division of the data set. We define a space as better dependent on the result of a function dependent on the size of the inner, outer and excluded partition. In our implementation we try different values for the Excluded Middle Partition so that it maximizes the function $f(i, o) = (i + o) - |i - o|$, where i is the size of the inner partition and o is the size of the outer partition. This is because we both want to minimize the size of the excluded partition, as well as balancing the the amount of objects in the inner and outer partition.

We also implemented hybrid pivot filtering, meaning a small increase in memory usage and a large decrease in the number of distance computations. Since we have to compute the distance to all pivots on each level anyway, the distance computation reductions for the objects come for free (disregarding extra CPU time used for pivot filtering).

3.4.2 Excluded Middle Vantage Point Forest

The EM-VP-Forest implementation is a hybrid implementation where the buckets use center objects from nodes as pivots in the buckets. If implemented naively, this would mean that the number of pivots is equal to the number of levels in the tree (which is maximum $\log_2(n)$). To restrict this, we only use the k last node centers as pivots. We have also used the the partition method as described in D-Index to divide the spaces better (in case the median is not a good choice).

We have modified the EMVP-Forest a bit from the original paper. Since the number of trees can be quite large (which makes indexing and range queries slow), we made a limit on the number of trees in a forest. We also gradually decrease ρ for each tree in the forest, to refrain from having a big tree with all the “leftovers” as the last tree in the forest.

3.4.3 SA-Tree

We implemented a variant of the HSA-tree[AMNR03], which again is a variation of the SA-tree. HSA is a tree that supports dynamic insertions (by using timestamps) and pivoting.

Since we did not need dynamic insertions(this project has focused on static structures), and it increased complexity of the implementation, we chose not to implement the dynamic part of the tree.

[AMNR03] presents two ways of selecting which objects should be pivots. The first uses the last n ancestors as pivots for an object, while the other use the best of all objects that the objects has been compared to. [AMNR03] shows that the latter method is only slightly better than the former, but requires more memory (because the reference of the best objects has to remembered as well as the distances). We chose to use the former since it was easier to implement and used roughly the same number of distance operations while using less memory.

3.4.4 Hierachy of clusters

HC was implemented as described in 2.4.2. To ease implementation of other cluster based methods, the implementation was done as open and extensible as possible.

First, how objects were saved in buckets were done so that several methods (AESA,LAESA) could be used. How clusters is split is also configurable, either by a constant number (LC) or with a function (HC and UHC). More leaves per node were also configurable (to help with PHC and UPHC).

3.4.5 GNAT

GNAT was implemented as described in 2.2.3. As with other implementations, hybrid indexing was implemented. With GNAT, this was done by letting the buckets being indexed with AESA. This limits the size of the buckets (since memory/CPU usage is squared with the number of objects in the bucket), but makes the method more effective.

3.4.6 M-Tree Based Methods

A few design decisions were made when implementing the M-Tree based methods. Since the Bonsai framework doesn’t really care about anything but the number of distance computations made, several of the features of the M-Tree makes little sense in our context. We have therefore eliminated some of these aspects where it allowed for a simpler implementation.

One of the differences is that our implementation is completely in-memory, rather than disk-based. The number of entries each node can accommodate, is set in the constructor, rather than being tweaked to the size of disk pages. This allows for more freedom in testing the impact different node sizes have on the M-Tree structures.

None of the implementations we made of M-Tree based structures take advantage of bulk-loading strategies for object insertion. Instead, we chose to insert each object one by one, as if we had no knowledge of how many would be inserted beforehand. Thus the results we get for construction time for M-Tree based methods may not be accurate in cases where you can insert several objects at once.

We have also chosen to not differentiate much between leaf nodes and internal nodes of the trees. Leaf node entries are stored in the same format as internal node entries, but with $r^c = 0$ and $ptr = null$. This allows us to limit the amount of special-case code to a minimum.

When performing a node split, we opted to distribute entries of split nodes into all new nodes, instead of letting the node being split keep some of its entries. The split node is then deleted from the tree. This is again done for simplicity purposes, as it has no impact on the number of distance calculations required. It is not however recommended for a disk-based approach.

Slim Tree

The Slim Tree is the only M-Tree based method that we let exploit the fact that we know how many objects are to be inserted. We do this by only letting the Slim-Down algorithm execute after all nodes have been inserted. This is because the Slim-Down algorithm requires a lot of distance computations to run, and is not meant to be executed after every single insert.

PM-Tree

We implemented the PM-Tree simply by subclassing the M-Tree, and adding HR arrays to the entries. Instead of using PD arrays in leaf node entries, we store the PD values as $HR[i].min = HR[i].max = PD[i]$. This way there is still no need to treat leaf nodes any different from internal nodes when pruning.

We also opted for using $n_{pd} = n_{hr}$. This choice was made because the benefits of using lower values of n_{pd} are seen in reduced I/O-costs rather than a reduced number of distance computations. Since this thesis is about comparing the number of distance computations different methods use, it seemed natural to use the configuration of each method that optimizes that metric.

M*-Tree and PM*-Tree

The M*-Tree and PM*-Tree were implemented by subclassing the M-Tree and PM-Tree respectively. Unfortunately our design didn't allow for the PM*-Tree to reuse the code for the M*-Tree (or vice versa), so we had to resort to a bit of dirty copy-paste coding to implement both of these trees. This caused a few bugs to need to be corrected twice, but the end result should still be good.

3.4.7 Efficient Metric MST Calculation

To be able to test our ideas on calculating Metric MST, we modified our Slim Tree implementation to allow alternate implementations for calculating MSTs. We implemented both our $O(n^3)$ and $O(n^2)$ algorithm.

The Slim Tree runs into a slight problem when not calculating all the distances however. After the Slim Tree has decided how to split the graph into two subgraphs, we need to locate suitable pivots for each of the two new nodes. The optimal pivot for a subgraph is the one minimizing the maximum distance to all the other nodes. When all the distances between nodes have already been calculated, this is easy to find, and requires no extra distance calculations. When all distances haven't been precalculated however, we cannot always be certain that we have found the optimal pivot without calculating extra distances. We came up with two possible solutions to this problem.

The first is to accept that we may not get an optimal pivot, and use the center of the subtree we get from splitting the MST in two. That is, we ignore the fact that the graph is more complex than the tree, and use the node that minimizes the maximum distance to all the other nodes, when only traversing edges that are part of the tree.

The second approach, is to use the distances we have calculated, and pick the node that has the minimum maximal distance as a temporary pivot. If this pivot has any uncalculated distances, we calculate them. If any of these distances caused the maximal distance to become larger than that one of the other nodes have, we pick the node that now is the "best", and repeat the procedure until no better nodes are found. This way of picking a pivot ensures that the best node is used.

We chose to go with the second approach, since we are guaranteed to be no worse off using our method than using that of the original Slim Tree. This should also help make testing construction costs of our splitting algorithm more directly comparable to construction costs of the Slim Tree, due to the resulting

Trees being almost identical (there can only be differences when two or more nodes are tied for being best pivots).

3.4.8 Using language constructs to implement hybrid methods

As mentioned in section 2.2.5 it is possible to implement a hybrid method with pivot filtering, where the pivots chosen are only objects where the distance have already been computed.

To implement this method, existing indexing methods may need to be modified to support filtering by pivots. This is not necessarily a simple task, and often the whole method has to be rewritten to facilitate the hybrid method. To help with this, we propose a new approach to implement pivot filtering in existing methods by using language constructs to make all distance computations lazy. This means that no modification has to be done to indexing methods.

We do this by wrapping the distance function in a separate class so that a call to the distance function does not return the actual distance, but a reference to a lazy evaluation object that represents the distance between the query and the metric object. For each object, a set of pivots are defined at indexing time (more about how these are chosen below). We then find the minimum and maximum distance to the object by looking at the distance to pivots already computed.

If the lazy object is compared with another lazy object, we find out if the minimum and maximum distance for each object and see if the their ranges overlap. If they don't, we can simply answer the comparison without actually computing the distance. If they do, we compute one of the objects and again check if they overlap.

In our scheme, arithmetic operations of distances are also supported. This is done by computing what the minimum and maximum distances will be, depending on the arithmetic operation (for example will the minimum of an addition of two objects be the sum of minimums of each object). If an arithmetic operation is asked to compute the distance, only one of the lazy objects will be computed, and the minimum and the maximum updated. This to refrain from doing unnecessary distance computations.

It is hard to make a definite rule to which objects should chosen as pivots. Although the method does not require that all pivot objects have to be computed, pivot objects that have not already been evaluated will only use up memory

One way of selecting pivot objects is to add all distance functions done during indexing with an object as a possible pivot, and then selecting the k

closest objects as pivot objects. This is a generally bad idea, since methods may have comparisons with objects that it is known will not help when doing search.

A small improvement is to add an object as an pivot only if it is the first argument to the distance function. For instantce, if $d(o_1, o_2)$ is computed during indexing, o_1 will be added as a pivot to o_2. The rationale behind this is that the function may be modified fairly easy, without changing how the function works without when our method is not used.

The last method is to let the method itself decide which objects should be pivots. This will give the best result (unless it is implemented in a bad way), but means that the method has to be modified to make use of the hybrid method.

Chapter 4

Results and Discussion

4.1 SSS-Tree

For the SSS-tree a variable α is used to define the arity of the tree. The authors of [BPS$^+$08] found that the best value of α was 0.4. To test if this really is the best value of the α variable, we did tests that varied α. As we see from 4.1 and 4.2, the optimal value of this variable varies from data set to data set. We also couldn't find that the optimal value would be 0.4. Our tests concluded with an optimal value near 0.2. This value does however have a considerably higher construction cost than higher values (because the tree is wider), so a value of 0.25 or 0.30 might be better to balance this off.

An explanation of this is that the measurement of α dates back to [BFPR06], which describes a method to find good pivots for pivoting methods. When used in a tree however, this value may be different.

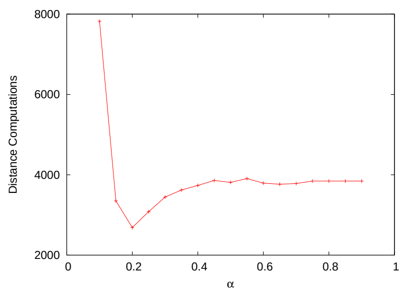


Figure 4.1: Distance operations on the Euclidean space

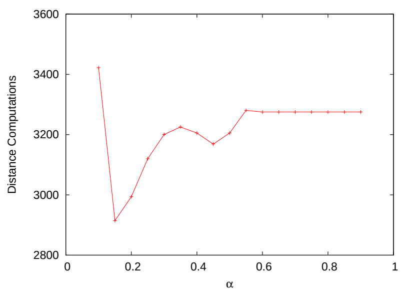


Figure 4.2: Distance operations on the clustered Euclidean space

There is also a lack of information of the testing procedures in [BPS$^+$08]. One example of this is that the paper mentions that nodes where the number of objects is less than δ should be put in a bucket. It is however not revealed what this value was when testing, making the findings hard to reproduce. There is also a lack of information about what parameters was used with the other trees tested.

4.2 Spatial Approximation Tree

We tested the method with different number of pivots to find the optimal number of pivots to use in the hybrid implementation. This was to see how well pivot filtering worked for the tree. The results can be seen in the figures 4.3 to 4.6. It is obvious that pivot filtering helps a lot, especially when no extra distance computations are needed during construction. At the same time we see that for the data sizes we tested, there is no need to go above 3 to 5 pivots, the exact value depending on the metric space.

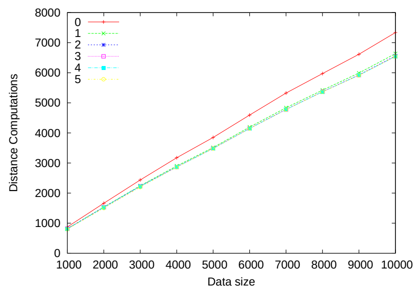


Figure 4.3: Distance operations with SA-Tree on the Euclidean space

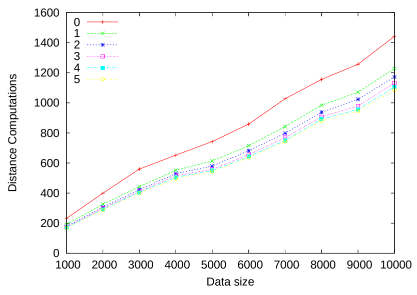


Figure 4.4: Distance operations with SA-Tree on the Clustered Euclidean space

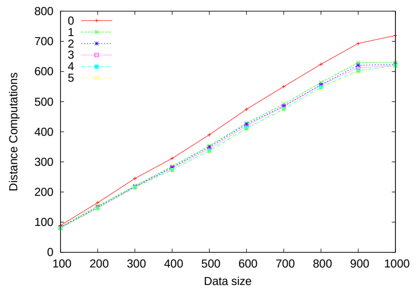


Figure 4.5: Distance operations with SA-Tree on QFD space

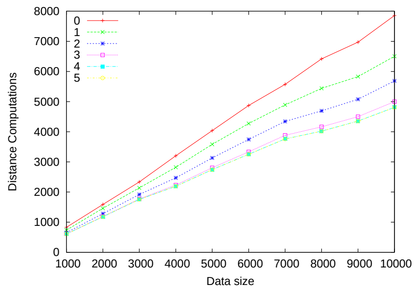


Figure 4.6: Distance operations with SA-Tree on the Edit Distance space

4.2.1 HC

To test HC/UHC, we tested three variations of HC. One was with $h(n) = \frac{n}{2}$, one with $h(n) = \frac{n^{1.0/1.4}}{2}$(UHC), and one with $h(n) = \frac{n}{3}$(UHC2). We also tested hybrid versions of each method, labeled AHC, UHC and UHC2. The result can be seen in figures 4.7 to 4.10. We chose not to test PHC or UPHC, since these are specialized methods for distances that can be computed using bit-parallel methods.

As can be seen from the results, the unbalanced methods are roughly as good as the balanced, sometimes better, sometimes worse. [Fre07] mentions that UHC may be better on metric spaces of higher dimensionality. We tested the methods with an Euclidean data set with 20 dimensions, see figure 4.11, which confirms the claim . UHC2 is however not as good, but still somewhat better than HC.

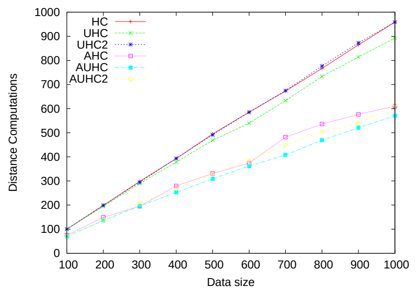


Figure 4.7: HC query costs with the Euclidean space

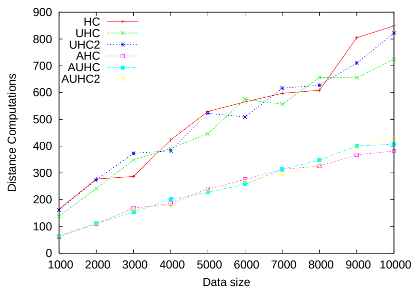


Figure 4.8: HC query costs with the clustered Euclidean space

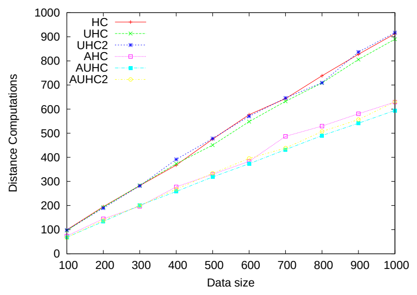


Figure 4.9: HC query costs with the QFD space

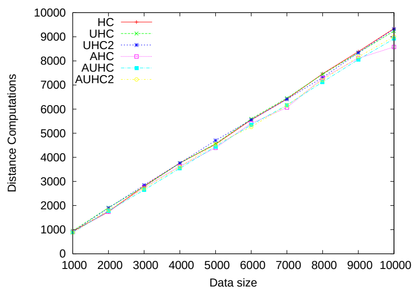


Figure 4.10: HC query costs with the Edit distance space

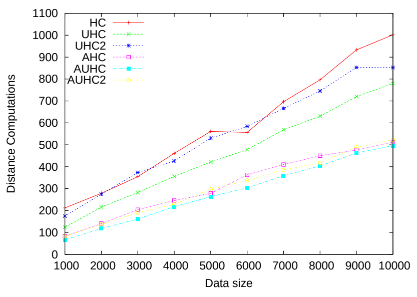


Figure 4.11: HC query costs with the Euclidean space - 20 dimensions

4.3 Excluded Middle Vantage Point Forest/D-Index

Since D-Index and EMVPF are two indexing methods that have a few similarities, we decided to test them together.

4.3.1 EMP-partition

As mentioned in Section 3.4.1, we implemented a small improvement to D-Index that should have helped with Excluded Middle-partitioning for certain data sets. After some testing we found out that the improvement was minuscule (the improvement, when present, was in the area of 0.3 %), and for some data sets it even increased the number of queries. Even for the data sets that saw improvements, it is fair to say that the extra complexity is not worth the gain.

4.3.2 D-Index vs EM-VP-Forest

When comparing D-Index with EM-VP-Forest (see figures 4.12 to 4.15), a few shortcomings with EM-VP-F surfaces that makes it less attractive.

First, EMVPF is harder to make hybrid. With D-Index, a constant number of objects is chosen as pivots, while with EMVPF, this is a variable number. When the search with EMVPF arrives at a bucket, it is unknown which pivots the objects in the bucket has already been compared with.

Second, since D-Index uses the same pivots for each level, distance operations inside the tree structure is lowered.

4.3.3 Language Constructs to Implement Hybrid Methods

As mentioned in section 3.4.8, we proposed to implement pivot filtering by using language constructs. The idea came quite late in the project, so we only man-

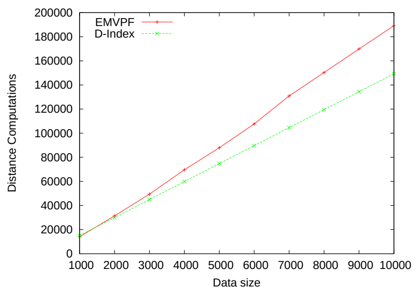


Figure 4.12: Distance operations(indexing) on the Euclidean space

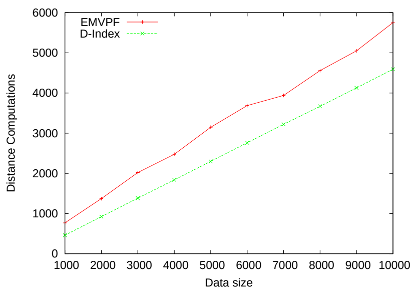


Figure 4.13: Distance operations(queries) on the Euclidean space

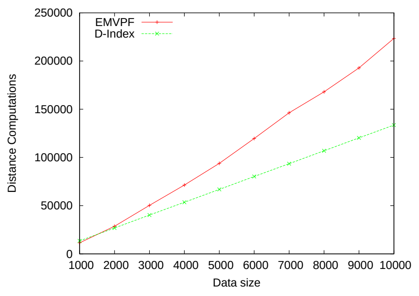


Figure 4.14: Distance operations(indexing) on the clustered Euclidean space

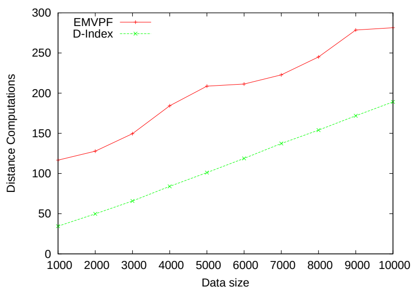


Figure 4.15: Distance operations(queries) on the clustered Euclidean space

aged to make a proof-of-concept implementation in Python that works on a few trees, which has a quite high overhead when doing distance computations. We have compared the SSS-tree with and without the language constructs method, and the result may be seen in Figure 4.16. The method clearly works (it is in any case guaranteed to not be worse in terms of distance computations), but it is unknown how it would compare to a custom-made hybrid implementation for SSS.

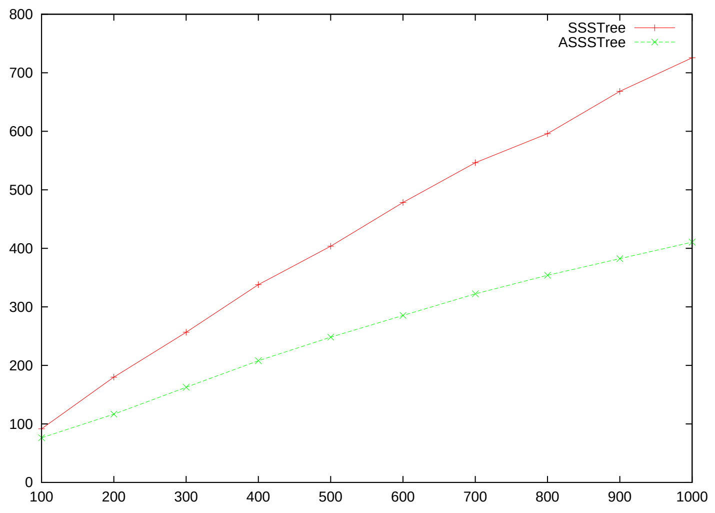


Figure 4.16: SSS-Tree with language constructs to implement pivoting

4.3.4 Tree Based Methods

We tested all the families of tree-based methods (except for EMVPF, since we tested it against D-Index) against each other to see which methods are the most promising. We tested with 1 % of the data set on each method. The results can be seen in 4.17 to 4.20. As shown, the different methods are good at different metrics. One outlier is the AUHC method, which is good at the clustered Euclidean space, but not that good on other methods. This can be explained by the fact that it is constructed to be especially good at clusters, being a variant of the LC (List-of-clusters) method.

4.4 PM-Tree Pivot Selection

The authors of the original paper [Sko04] didn't say anything about how one should select global pivots for the PM-Tree. Therefore, we decided to implement two different strategies for pivot selection, and test them to see which is the best.

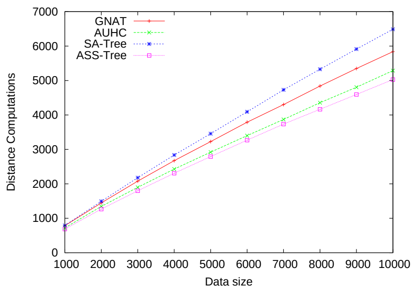


Figure 4.17: Distance operations for Tree Based methods on the Euclidean space

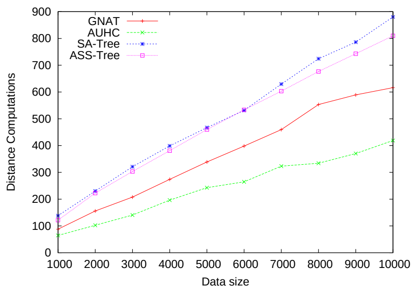


Figure 4.18: Distance operations for Tree Based methods on the Clustered Euclidean space

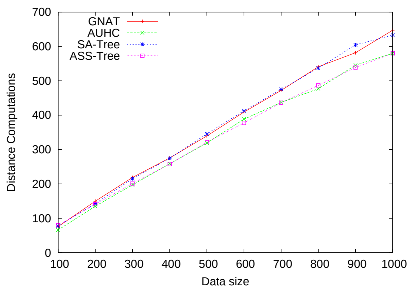


Figure 4.19: Distance operations for Tree Based methods on QFD space

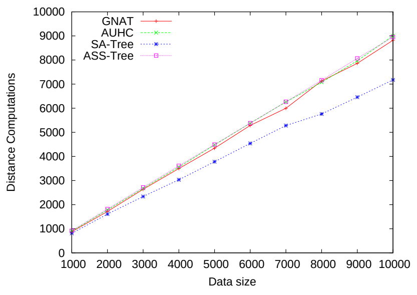


Figure 4.20: Distance operations for Tree Based methods the Edit Distance space

The two strategies we implemented are "completely random", and Bustos (see section 2.1.2).

We tested the two methods using 20 pivots against both the Euclidean and Clustered Euclidean distance metrics. The results can be found in figures 4.21 and 4.22, and clearly show that the Bustos method of pivot selection outperform random selection. We will therefore use the Bustos method in all other tests of the PM-Tree structure.

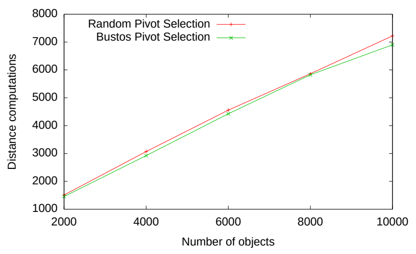


Figure 4.21: PM-Tree query costs with the Euclidean distance metric

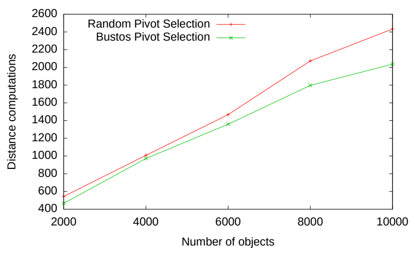


Figure 4.22: PM-Tree query costs with the Clustered Euclidean distance metric

4.5 Number of Pivots in the PM-Tree

It is desirable to know how the number of pivots affect the search efficiency of the PM-Tree. Obviously the more pivots, the less distance computations will be necessary when searching. However, every pivot adds both to construction costs and memory requirements.

We tested the PM-Tree with various number of pivots, and the results can be seen in figures 4.23 through 4.26. When measuring construction costs, 10 different samples were used for each test. The best and worst result were removed, and the displayed result is the average of the remaining 8 samples. For query costs, the results displayed are the average over 30 queries, distributed evenly over 3 different sample data sets for every data point.

The results seen indicate that construction costs scale linearly with increasing number of pivots. This is not especially unexpected, as each extra pivot requires a distance calculation from every object inserted into the tree.

Query costs seems to drop by about the same amount every time the number of pivots doubles. This makes it hard to set a firm recommendation on how many pivots to use. Both construction costs and memory requirements scale linearly with the number of pivots, so the optimal number of pivots is both dependant on the type of problem the tree is being built to solve, as well as the system resources available.

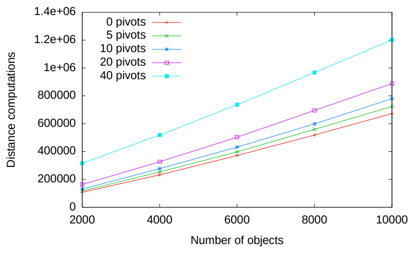


Figure 4.23: PM-Tree construction costs with the Euclidean distance metric

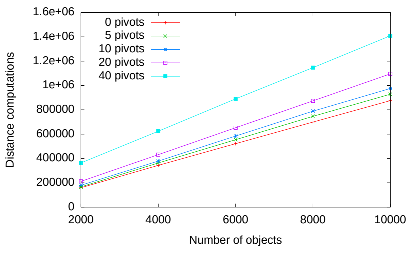


Figure 4.24: PM-Tree construction costs with the Clustered Euclidean distance metric

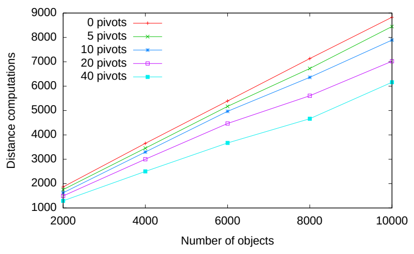


Figure 4.25: PM-Tree query costs with the Euclidean distance metric

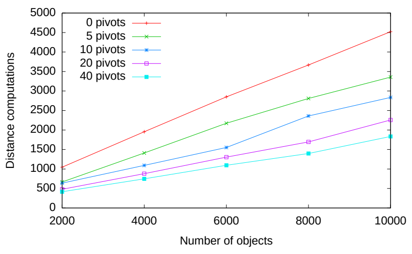


Figure 4.26: PM-Tree query costs with the Clustered Euclidean distance metric

We chose to use 20 pivots for the other tests involving the PM-Tree. This is partly because it seems like a reasonable compromise between query performance and memory requirements, and partly because it matches the number of pivots we use in the LAESA method.

4.6 Efficient Metric MST Calculation

Our tests of our $O(n^2)$ MST algorithm showed very little improvement over the standard MST algorithm. The number of distance computations were almost identical to that of the Slim Tree. We have therefore omitted plotting the results, and only compare our $O(n^3)$ algorithm to the standard one.

For each test, 10 random data sets were used. The highest and lowest construction costs were dropped, and the results displayed are the averages over the remaining 8 data sets.

The results can be seen in figures 4.27 through 4.30. “Standard” refers to the original Slim Tree implementation, and “Improved” is when the MST algorithm has been replaced by our $O(n^3)$ algorithm. The numbers represent the number of entries in each node, so “Standard, 64” denotes a standard Slim Tree with up to 64 entries in each node.

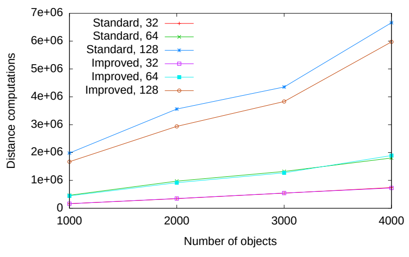


Figure 4.27: Slim Tree construction costs with the Euclidean distance metric

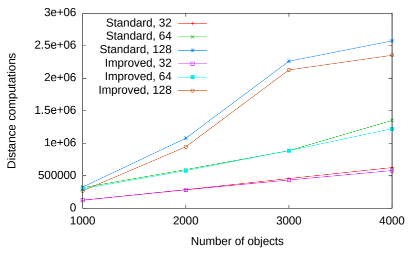


Figure 4.28: Slim Tree construction costs with the Clustered Euclidean distance metric

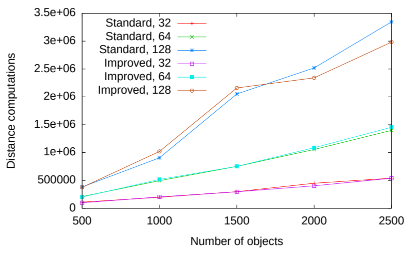


Figure 4.29: Slim Tree construction costs with the Edit Distance distance metric

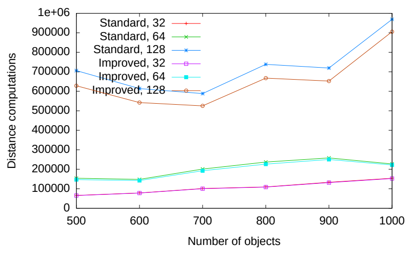


Figure 4.30: Slim Tree construction costs with the QFD distance metric

As we can see, the improved MST algorithm result in lower construction costs. We can also see that a higher number of objects, and a higher number of objects per node are both qualities that improves our algorithm's performance relative to the standard MST algorithm.

The only instance of the standard MST algorithm outperforming ours is when using the Edit Distance distance metric. There we see a higher construction cost for the data sets with 1000 and 1500 objects. However, these cases can most likely be attributed to the nature of the distance metric; since the distances between objects are discrete, it is more likely that more than one object is the "best" candidate as a pivot for a node. This will result in the two methods to be more likely to result in different trees, which has potentially large random effects on construction costs.

Since the original M-Tree uses a $O(n^3)$ splitting algorithm, we don't think that our algorithm's higher complexity is too big of an obstacle. Our suggestion is to choose which algorithm to use depending on the complexity of distance calculations. If distance computations are expensive, use our algorithm, otherwise, stick with the standard one.

4.7 M-Tree Based Methods

In order to get a good overview of how the different M-Tree based methods compare to each other, we tested each of them using all our four distance metrics. Construction costs were measured using 10 data sets for each number of objects. The same sets were then indexed by each method, and the number of distance computations counted. The highest and lowest count for each method were dropped, and the results can be seen in figures 4.31 through 4.34.

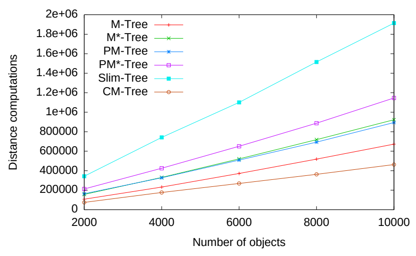


Figure 4.31: M-Tree Based construction costs with the Euclidian distance metric

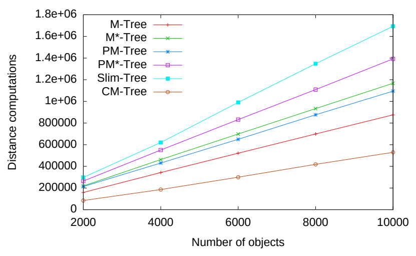


Figure 4.32: M-Tree Based construction costs with the Clustered Euclidian distance metric

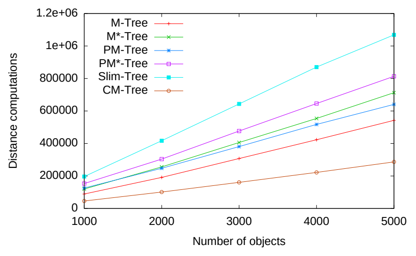


Figure 4.33: M-Tree Based construction costs with the Edit Distance distance metric

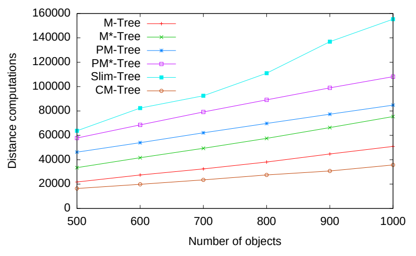


Figure 4.34: M-Tree Based construction costs with the QFD distance metric

The results seem pretty clear. The CM-Tree has by far the lowest construcion costs in the M-Tree family. The Slim Tree has the largest costs. Between these extremes we have the M-Tree, PM-Tree, M*-Tree and PM*-Tree, in that order. The only exception to this is with the QFD distance metric, where the M*-Tree and the PM-Tree have swapped places.

The reason for the low construction costs of the CM-Tree is probably due to cheap node splits. All the other methods need $O(n^2)$ distance computations to perform a node split, but the CM-Tree needs only $O(n)$. This difference would most likely be even bigger if a larger node size had been used.

To measure query costs, ten range queries were issued on three different datasets of each size. The results can be found in figures 4.35 through 4.38.

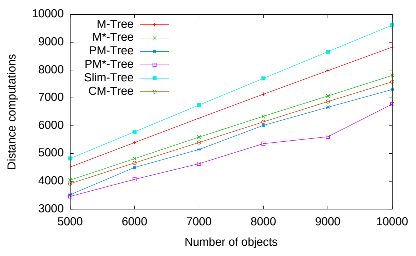


Figure 4.35: M-Tree Based query costs with the Euclidian distance metric

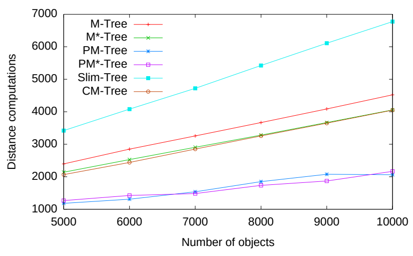


Figure 4.36: M-Tree Based query costs with the Clustered Euclidian distance metric

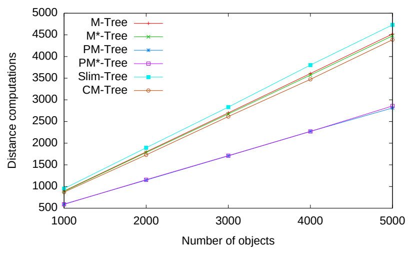


Figure 4.37: M-Tree Based query costs with the Edit Distance distance metric

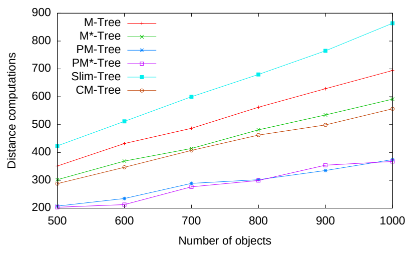


Figure 4.38: M-Tree Based query costs with the QFD distance metric

The query results are a bit more varied than the construction results. Still, the PM-Tree and PM*-Tree stand out as clear winners. They are roughly tied for first place in all distance metrics except the Euclidian one.

When using the Euclidian distance metric, the PM*-Tree beats the PM-Tree by a wide margin. We can also see that in this distance metric, the gap between the PM-Tree and the M*-Tree drastically reduces. The reason for these results is most likely due to the non-clustered nature of the metric. In the other metrics, it is easier to pick global pivots in a way that makes sure most of the objects has a reasonably near pivot. In a completely random vector space this is harder. However, the Nearest-Neighbor Graphs that the M*-Tree and PM*-Tree uses will still provide good pivots for most objects. Therefore these methods shine when there is no clustering effects.

4.8 Comparison of The Most Promising Methods

To test each group of methods against each other, we chose the best methods from each group and ran against our data sets. We also included the LAESA method (developed in [FF07]), so that we can test our newly developed methods against another. The results can be found in 4.39 to 4.46.

The results of this last comparison of methods are however not as useful as we would like them to be. This is because the number of distance computations alone is not the only way to describe performance of an indexing methods. The amount of I/O and memory usage and computation overhead has not been addressed.

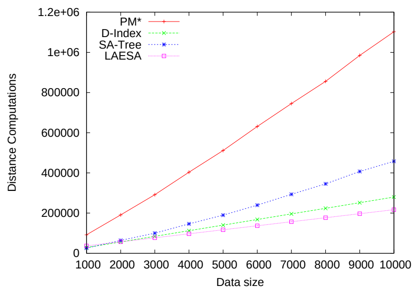


Figure 4.39: Distance operations (indexing) on the Euclidean space

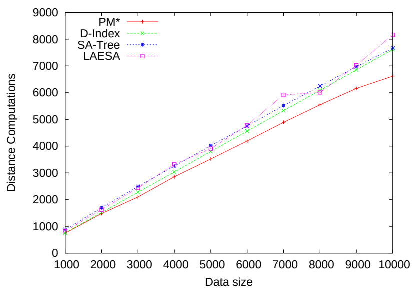


Figure 4.40: Distance operations (queries) on the Euclidean space

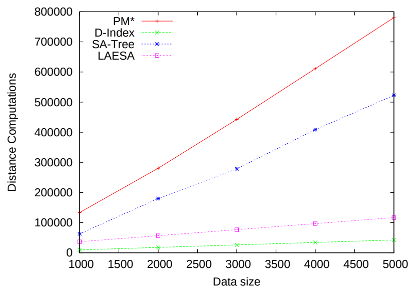


Figure 4.41: Distance operations (indexing) on Edit Distance space

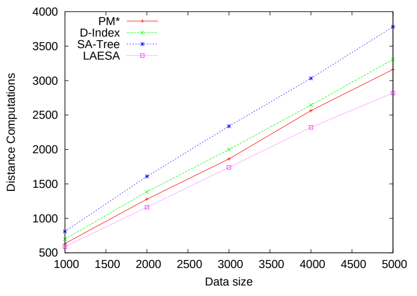


Figure 4.42: Distance operations (queries) on the Edit Distance space

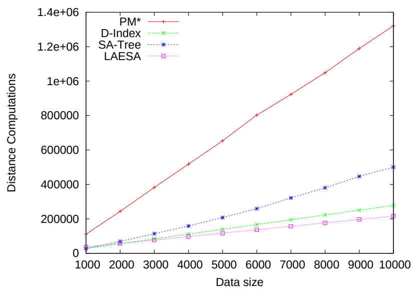


Figure 4.43: Distance operations (indexing) on the Clustered Euclidean space

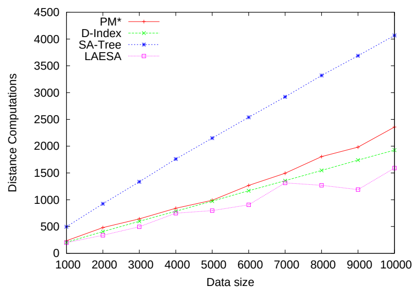


Figure 4.44: Distance operations (queries) on the Clustered Euclidean space

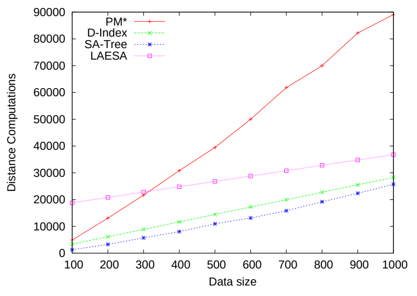


Figure 4.45: Distance operations (indexing) on the QFD space

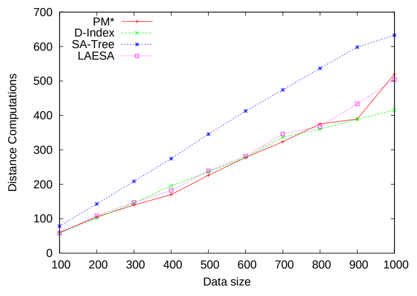


Figure 4.46: Distance operations (queries) on the QFD space

Chapter 5

Conclusion

During this thesis, we have implemented methods that have been proposed in newer literature. The number of methods in this field is very high, and many methods do not have a right of life. It is an important task to filter out the methods that need further study.

Testing methods on different metric spaces show that the properties of the metric space used can have a large impact on the results seen. This is an important observation, since an euclidean metric space (which is the space most often chosen to test metric indexing methods) may show results that are different from what would be experienced in a real-life metric space.

Our experiments show that hybrid methods are| superior to non-hybrid methods. When using pivoting also for inner nodes, the results are even better. There is an overhead when using pivoting, but this is not very large when compared to the benefits.

Out of the methods we tested, the CM-Tree is by far the best of the ones not employing global pivoting. We have also seen that adding global pivots to existing methods generally boost their results significantly. From this we conclude that the CM-Tree with an addition of global pivots is likely to perform very well.

Chapter 6

Future work

Although we have implemented many methods through this project, the Bonsai library is still not complete. Several methods could also be studied and implemented further, to see if the number of distance computations can be reduced.

In this paper, we propose a method to implement pivot selection by using programming language constructs. Since this method was only implemented as a proof-of-concept quite late in the thesis, it was not tested with many methods. Further study of this concept would be interesting.

When benchmarking, we have concentrated on the number of distance computations. However, many of the metric indexing functions are geared towards reducing I/O - operations. It would be interesting to develop test cases that are geared against these types of indexing methods.

The CM-Tree seems like a very competitive version of the M-Tree, but it lacks global pivoting. Introducing pivots to CM-Tree seems like a promising concept, and we belive it may prove better than any of the methods we looked at during our work.

Bibliography

[AMNR03] Diego Arroyuelo, Francisca Muñoz, Gonzalo Navarro, and Nora Reyes. Memory-adaptative dynamic spatial approximation trees. In *Proceedings of the 10th International Symposium on String Processing and Information Retrieval, SPIRE*, 2003.

[AS07] Lior Aronovich and Israel Spiegler. Efficient similarity search in metric databases using the CM-tree. *Data & Knowledge Engineering*, 2007.

[BFPR06] Nieves R. Brisaboa, Antonio Farina, Oscar Pedreira, and Nora Reyes. Similarity search using sparse pivots for efficient multimedia information retrieval. In *ISM '06: Proceedings of the Eighth IEEE International Symposium on Multimedia*, pages 881–888, Washington, DC, USA, 2006. IEEE Computer Society.

[bib98] The bible, douay-rheims version. `http://www.gutenberg.org/etext/1581`, 1998.

[BNC01] Benjamin Bustos, Gonzalo Navarro, and Edgar Chavez. Pivot selection techniques for proximity searching in metric spaces. *In Proc. of the XXI Conference of the Chilean Computer Science Society*, 2001.

[BPS$^+$08] Nieves Brisaboa1, Oscar Pedreira1, Diego Seco1, Roberto Solar, and Roberto Uribe. Clustering-based similarity search in metric spaces with sparse spatial centers. *Lecture Notes in Computer Science*, pages 186–197, 2008.

[Bri95] Sergey Brin. Near neighbor search in large metric spaces. In Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio, editors, *VLDB '95: proceedings of the 21st International Conference on Very Large Data Bases, Zurich, Switzerland, Sept. 11–15, 1995*, pages 574–584, Los Altos, CA 94022, USA, 1995. Morgan Kaufmann Publishers.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, chapter 23. The MIT Press, second edition, 2001.

[CN05] E. Chávez and G. Navarro. A compact space decomposition for effective metric indexing. *Pattern Recognition Letters*, 26(9):1363–1376, 2005.

[CNBYM01] E. Chávez, G. Navarro, R. Baeza-Yates, and J.L. Marroquin. Searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, September 2001.

[Com79] Douglas Comer. Ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, 1979.

[CPZ97] Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree: An efficient access method for similarity search in metric spaces. In Matthias Jarke, Michael J. Carey, Klaus R. Dittrich, Frederick H. Lochovsky, Pericles Loucopoulos, and Manfred A. Jeusfeld, editors, *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, pages 426–435. Morgan Kaufmann, 1997.

[CTTSF00] Jr. Caetano Traina, Agma J. M. Traina, Bernhard Seeger, and Christos Faloutsos. Slim-trees: High performance metric trees minimizing overlap between nodes. In *EDBT '00: Proceedings of the 7th International Conference on Extending Database Technology*, pages 51–65, London, UK, 2000. Springer-Verlag.

[DGSZ03] Vlastislav Dohnal, Claudio Gennaro, Pasquale Savino, and Pavel Zezula. D-index: Distance searching index for metric data sets. *Multimedia Tools Appl.*, 21(1):9–33, 2003.

[FF07] Rune Fevang and Arne Bergene Fossaa. Metric indexing - implementation and benchmarking of selected methods. `http://home.orakel.ntnu.no/~arnebef/project.pdf`, 2007.

[Fre07] Kimmo Fredriksson. Engineering efficient metric indexes. *Pattern Recogn. Lett.*, 28(1):75–84, 2007.

[GSZ01] Claudio Gennaro, Pasquale Savino, and Pavel Zezula. Similarity search in metric databases through hashing. In *MULTIMEDIA '01: Proceedings of the 2001 ACM workshops on Multimedia*, pages 1–5, New York, NY, USA, 2001. ACM.

[Gut84] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In Beatrice Yormark, editor, *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, June 18-21, 1984*, pages 47–57. ACM Press, 1984.

[MHGM05] Pierre-Alain Moëllic, Patrick Hède, Gregory Grefenstette, and Christophe Millet. Evaluating content based image retrieval techniques with the one million images clic testbed. In Cemal Ardil, editor, *WEC (2)*, pages 171–174. Enformatika, Çanakkale, Turkey, 2005.

[MOC96] Luisa Mico, Jose Oncina, and Rafael Carrasco. A fast branch and bound nearest neighbour classifier in metric spaces. *Pattern Recognition Letters archive*, 1996.

[Nav99] Gonzalo Navarro. Searching in metric spaces by spatial approximation. In *SPIRE/CRIWG*, pages 141–148, 1999.

[NBE$^+$93] Carlton W. Niblack, Ron Barber, Will Equitz, Myron D. Flickner, Eduardo H. Glasman, Dragutin Petkovic, Peter Yanker, Christos Faloutsos, and Gabriel Taubin. Qbic project: querying images by content, using color, texture, and shape. *Storage and Retrieval for Image and Video Databases*, 1908(1):173–187, 1993.

[NR01] Gonzalo Navarro and Nora Reyes. Dynamic spatial approximation trees. In *Proceedings of the XXI Conference of the Chilean Computer Science Society, SCCC*, pages 213–222, 2001.

[SH07] T. Skopal and D. Hoksza. Improving the performance of m-tree family by nearest-neighbor graphs. In *Proceedings of the Eleventh East-European Conference on Advances in Databases and Information Systems, ADBIS*, 2007.

[Sko04] Tomáš Skopal. Pivoting M-tree: A metric access method for efficient similarity search. In V. Snášel, J. Pokorn'y, and K. Richta, editors, *Proceedings of the Annual International Workshop on DAtabases, TExts, Specifications and Objects (DATESO)*, volume 98 of *CEUR Workshop Proceedings*, Desna, Czech Republic, April 2004. Technical University of Aachen (RWTH).

[Vid86] Enrique Vidal. An algorithm for finding nearest neighbours in (approximately) constant average time. *Pattern Recognition Letters*, 1986.

[Vil85] Juan Miguel Vilar. Reducing the overhead of the aesa metric-space nearest neighbour searching algorithm. *Information Processing Letters*, 1985.

[Yia98] Peter N. Yianilos. Excluded middle vantage point forests for nearest neighbor search. Technical report, NEC Research Institute, Princeton, NJ, July 1998.

[ZADB05] Pavel Zezula, Giuseppe Amato, Vlastislav Dohnal, and Michal Batko. *Similarity Search: The Metric Space Approach (Advances in Database Systems).* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.