Signe Marie Øen Carlsen
Helene Haukås Moe

Similarity Search in Metric Spaces with Weighted Multi-Focal Regions

Using the Ambit Region Type to Improve the Performance of the SSS-Tree

Master's thesis in Computer Science
Supervisor: Magnus Lie Hetland
June 2020

Signe Marie Øen Carlsen
Helene Haukås Moe

Similarity Search in Metric Spaces with Weighted Multi-Focal Regions

Using the Ambit Region Type to Improve the Performance of the SSS-Tree

Master's thesis in Computer Science
Supervisor: Magnus Lie Hetland
June 2020

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Kunnskap for en bedre verden

Abstract

Similarity searching through metric indexing is an essential tool for retrieving data from large data sets. When creating metric indices, clustering can be used. Most existing clustering-based metric indexing methods, including the SSS-tree, use regions with one focus only. We tested the performance of a modified version of the SSS-tree, named the WSSS-tree, which uses multi-focal regions, so-called ambits, trained to avoid a set of example queries. The WSSS-tree significantly outperformed the SSS-tree when applied to queries from one cluster, and had a slight advantage when applied to queries from two clusters. For queries from two clusters, we also tested a modified version of the WSSS-tree, named W2SSS-tree, which significantly outperformed both the WSSS-tree and SSS-tree. These results suggest that using the ambit region type with multiple weighted foci in metric indices can reduce the number of distance calculations needed to search for a query if one knows the distribution that the queries will come from.

Sammendrag

Metrisk indeksering er et nyttig verktøy for å søke i store datamengder. For å lage metriske indekser, kan man dele datamengden inn i klynger. De fleste eksisterende klyngebaserte metriske indekseringsmetoder, blant disse SSS-treet, bruker regioner med bare ett fokus. Vi testet ytelsen til en modifisert versjon av SSS-treet, kalt WSSS-treet, som bruker regioner med flere fokus (*ambits*). Formen til disse regionene er laget ved å trene dem til å unngå et sett med eksempelspørringer ved å gi fokusene ulik vekting. WSSS-treet presterte vesentlig bedre enn SSS-treet på spørringer fra én klynge, og noe bedre på spørringer fra to klynger. På spørringer fra to klynger testet vi også en modifisert versjon av WSSS-treet, kalt W2SSS-treet, som presterte vesentlig bedre enn både SSS-treet og WSSS-treet. Disse resultatene tyder på at å bruke regioner med flere vektete fokus (*ambits*) kan redusere antall avstandssammenligninger man trenger for å utføre en spørring hvis man på forhånd kan vite noe om fordelingen spørringene vil komme fra.

Contents

Chapter 1

Introduction

In the growing age of multimedia, similarity searching is an essential tool when retrieving data from large data collections. To perform such searches, one needs a measure of the similarity between objects, which can be modeled by a distance function. This distance function is often expensive to compute, and the number of distance calculations is considered the query cost.

To minimize the cost of conducting queries, one can build an index on the database before searching it. A way of creating these indices is to use *metric indexing*, which translates complex data relations to a *metric space*, where all points have one real, symmetric distance to each other, and the triangle inequality is satisfied [14].

The SSS-tree [5] is a clustering-based metric indexing method, which uses SSS (Sparse Spatial Selection) to create the cluster centers in the index. The method adapts the selected cluster centers to the intrinsic dimensionality of the space so that they will be well distributed in space. The SSS-tree has been shown to outperform other well-known clustering-based indexing methods such as M-Tree [10], GNAT [4], and EGNAT [28].

However, the SSS-tree regions have only one cluster center, or focus, restricting the regions to be ball-shaped. Hetland introduces the ambit region type [17], which generalizes the description of regions to allow for a wide range of shapes by, for example, allowing multiple foci, which can be weighted differently. There is reason to believe that an index that allows regions to be of multiple different shapes could be more efficient than an index that restricts its regions to be of only one shape [17].

When searching an SSS-tree, if a node's region overlaps with the query, all of its children need to be examined. Therefore, a node can have all of its sibling nodes as foci in its region, without this contributing to more distance calculations. By varying the weights of these foci, there are vast possibilities for the shape of each region.

The goal of this project is to implement and empirically assess the performance of an index that is an adaptation of the SSS-tree that uses the ambit region type, allowing each region to have all its siblings as foci. We will call such an index a WSSS-tree. To find the ideal shape of each region, one can perform a kind of training using example queries from a distribution of queries that one thinks the index will be applied to. The weights of the foci in each region will be optimized so that the region will be shaped to avoid overlapping with the average of these *training queries*.

We will answer the following research question in this report: *Can the WSSS-tree outperform the SSS-tree? If so, in which scenarios will it have the most significant advantage?*

A vector containing the average distance from the training queries to each focus is used when deciding the weighting of the foci. Therefore, this average feature vector needs to be representative of all training queries for beneficial optimizations of the regions. This seems to be the case if the queries come from one cluster, and from this emerged hypotheses 1 and 2:

- **Hypothesis 1** The WSSS-tree will outperform the SSS-tree if the queries come from one cluster.

- **Hypothesis 2** The WSSS-tree will lose a part of its advantage over the SSS-tree when the queries come from more than one cluster if the clusters are so far apart that they would not be classified as one larger cluster.

These hypotheses were tested in experiments 1 and 2 in this report, by applying queries from one and two clusters, respectively, to the indices. A combination of the WSSS-tree and the SSS-tree, called a CSSS-tree, was used as a lower bound in these experiments. The results showed that the WSSS-tree outperformed the SSS-tree when applied to queries from one cluster, but lost part of its advantage when applied to queries from two clusters. The performance of the WSSS-tree was always close to the performance of the CSSS-tree. For the WSSS-tree to regain its advantage over the SSS-tree, we tested a new index structure: the W2SSS-tree, a modified version of the WSSS-tree that uses two facets, one for each cluster of training queries. Following this, Hypothesis 3 was formed:

- **Hypothesis 3** The W2SSS-tree will outperform both the SSS-tree and the WSSS-tree when the queries come from two clusters if the clusters are so far apart that they would not be classified as one larger cluster.

This hypothesis was tested in Experiment 3 in this report. The W2SSS-tree performed better than both the SSS-tree and WSSS-tree when applied to queries from two clusters.

We believe that when the queries come from more than two clusters, a weighted tree will still perform better than an SSS-tree if the tree has as many facets as there are query clusters. However, for this project, we only used queries from one and two clusters.

Chapter 2

Background and Basic Concepts

One of the challenges of working with large data sets is to retrieve a desired part of the data as efficiently as possible. Similarity search and metric indexing can be tools one can use to achieve this. In this chapter, we will cover the basics of similarity search and metric indexing, and give an introduction of the SSS-tree: the metric index structure that the new indexing methods presented later in this report will be based on.

2.1 Similarity Search

For many data types, among these multimedia data, the similarity between the objects can be the only measure of comparison one can apply to the data. Therefore, having a way to search for data that is similar to a particular object is necessary.

A similarity search problem involves finding the object(s) that is (are) the most similar to a query in a collection of objects. These objects, as well as the query, are often characterized by a collection of relevant features and represented as points in a high-dimensional attribute space [13]. However, similarity search can also be applied to objects in other spaces, such as string edit distance.

2.1.1 Motivating Applications of Similarity Search

Traditionally, search operations have been applied to structured data, such as numerical or alphabetical information. In these applications, the search result retrieved is precisely equal to the search query. Traditional databases are built to search for records that match a search key exactly. More advanced searches, such as range queries on numerical keys, still rely on there being a total linear order on the keys [7].

Collections of multimedia data such as images, audio, and video, have emerged with the evolution of information and communication technologies. Such information is not naturally structured and can not be queried through key-matching, meaning that the data is neither ordered nor can be compared for equality [7].

Furthermore, in multimedia applications, one is usually not interested in searching for segments precisely equal to each other. Instead, retrieving objects similar to a given object is what is of interest. An example of a query that could be of interest is "Find an image of a red car on the highway." This could be easily solved using a classical database if the repository is tagged with a full description of what is inside the image. However, image recognition technology is still not able to tag image motives as a human would. An alternative would be to provide an example image and return images similar to this. Such a problem could be classified as a similarity search problem, where one would have defined a distance function for the similarity between the image objects represented by a selection of the images' features [7].

Another application of similarity search is spelling. When searching with a correctly spelled query, documents containing a misspelled word could not be retrieved using a classical searching scheme. In this case, we want to retrieve all words that are close to the query. Related to this is the application of spell checkers, where we want to look for close variants of misspelled words. This could be done by conducting a similarity search using the string edit distance between the words as the distance function [7].

2.2 Metric Indexing

One can abstract the similarity between objects to a metric space. The similarity/dissimilarity between objects in a metric space is described by the distance between them.

2.2.1 Metric Search Indexing Problem

In this report, we will use the following definition of a metric search indexing problem [30]:

Definition 1 *Metric Search Indexing Problem. Let D be a domain, d a distance measure on D, and (D, d) a metric space. As the space is metric, the following holds: positiveness (distance $d(x, y) \geq 0$, $d(x, x) = 0$), symmetry ($d(x, y) = d(y, x)$), and the triangle inequality ($d(x, y) + d(y, z) \geq d(x, z)$). Given a set $X \subseteq D$ of n elements, preprocess the data so that similarity queries are answered efficiently.*

2.2.2 Types of Queries

The queries in a metric search indexing problem can be of different types. The most basic kind is a *similarity range query*, where all objects within a certain radius of the query object are of interest [15]. An example of a range query is found in Figure 2.1. A potential issue with this type of query is that it will return an unpredictable number of data objects.

Another query type is a k-NN-query, where the k nearest neighbor to a query object is retrieved. An example of a k-NN-query with $k = 4$ is found in Figure 2.2. If only the closest neighbor is of interest, the query is called an NN-query [7]. Applying k-NN-queries is more complicated than applying range queries and can be executed through a series of range queries. This can be done by first executing a range query where the query object is the same as in the k-NN-query, but the radius is infinite, so all objects in the index will overlap with the query. The first k objects to overlap with the query will be added to a set of possible candidates to be the result. Next, a range query with the radius set as the distance from the query object to the candidate object furthest away is applied. Every object that overlaps this range query will be swapped with the candidate object that is furthest away from the query. This process is repeated until there are no new overlaps, and at this point, the set of candidates will contain the query object's k nearest neighbors.

A way to simulate performing a range query with a predictable number of results like a k-NN-query without the complexity of performing a k-NN-query, is to sort the data set with respect to the distance to the query object and apply a range query with the radius equal to the distance from the query object to the kth element of the sorted list [16]. However, this is not an efficient way to apply a query and is only applicable for simulation used in research.

If one is not dependent on predictably retrieving an exact number of objects, a simpler approach would be to apply range queries with different radii and select the radius value that returns approximately the number of objects desired. This is what we did for the experiments in this report.

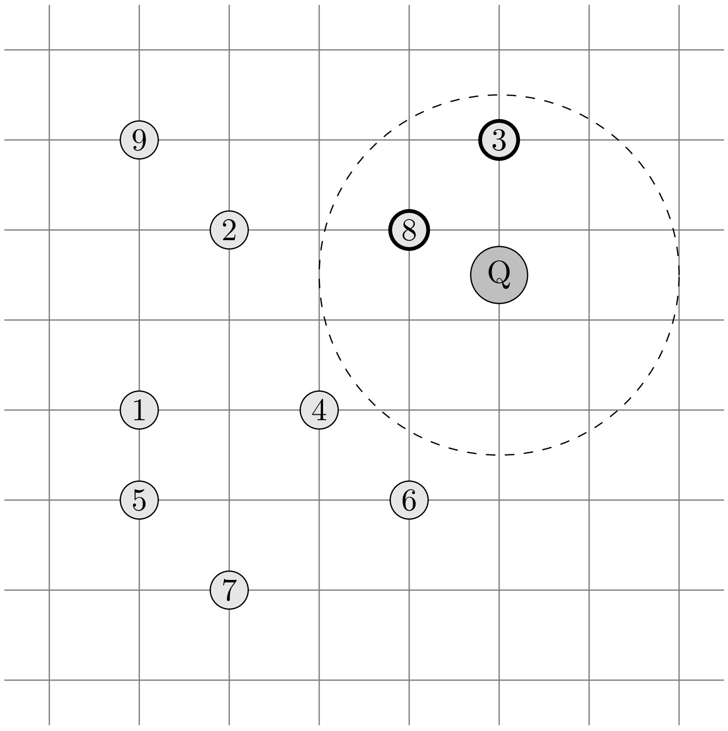


Figure 2.1: An example of a range query. Q is the query object, and the dashed circle marks the query radius. Data object 3 and 8 will be the result of the range query as these lie within the query radius.

2.2.3 Metric Indexing Methods

Metric indexing methods can be classified into two categories: pivot-based methods and clustering-based methods. Pivot-based indexing methods calculate and store the distance from each object in the data set to a subset of objects that are chosen as pivots. These pre-calculated distances are used to prune the data set when searching instead of comparing each object to the query [8]. AESA is an example of an efficient pivot-based method, but since every object can play the role of a pivot, it takes up much space in memory [29].

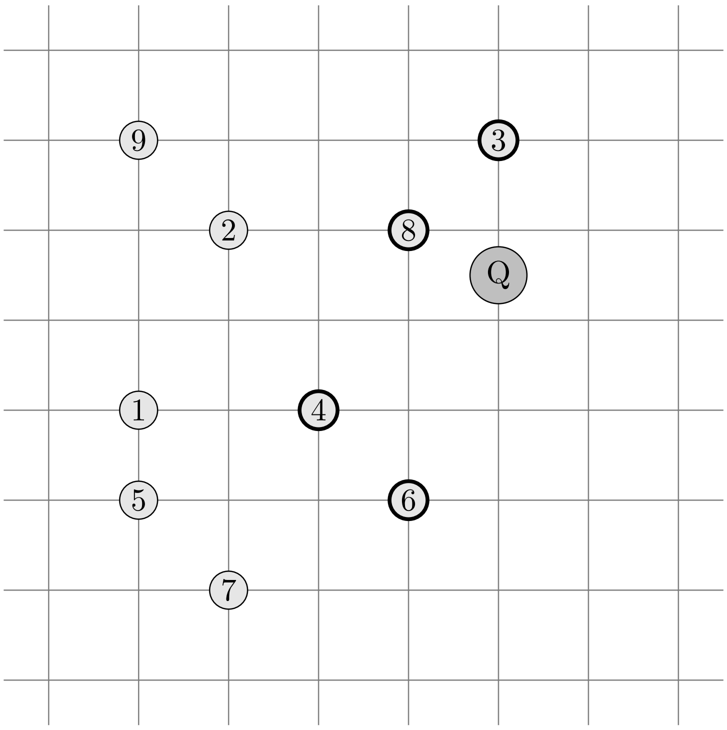


Figure 2.2: An example of a 4-NN-query. Q is the query object, and 3, 4, 6, and 8 are the four closest data objects to Q, and will be the result of the 4-NN-query.

Clustering-based indexing methods pre-process the data set to a set of clusters, or regions. The information about the regions is stored in the index. When searching for a query in such an index, one can discard entire regions from the result. Examples of well-known clustering-based indexing methods are SSS-tree [5], M-Tree [10], GNAT [4], and EGNAT [28].

2.2.4 The Ambit Region Type

The indices used in this report are clustering-based, which means that the information used for searching is stored in the form of regions.

To generalize the description of regions, the Ambit Region type was introduced by Hetland [17]:

Definition 2 *Ambit [17]. An ambit of degree m in a universe U with sym-*

metric comparison function $\delta : U \times U \to K$ is a comparison-based region

$$B[p, r; f] := C[p, x : f(x) \leq r] \tag{2.1}$$

as defined by

(i) a tuple p of sources or foci $p_1, ..., p_m \in U$;

(ii) a radius $r \in L$; and

(iii) a remoteness map $f : K^m \to L$, with L partially ordered.

The features x_i of u are called its radients, and $f(x)$ is its remoteness. The ambits are linear if the remoteness map f is linear.

The shape of an ambit is determined by its foci and its remoteness map. The radius decides the size of the ambit. In figs. 2.3 to 2.5, the remoteness map is the sum of the radients, but the number of foci varies from one to three, which creates a ball, an ellipse, and an egglipse respectively. The foci are weighted equally in all three figures.

In Figure 2.6, the foci of the ambit are weighted differently, creating an ambit that resembles an egglipse more than an ellipse, even though it has two foci, not three.

The ambit in Figure 2.7 has the remoteness map $x_1 - x_2$, and forms a hyperboloid. Figure 2.8 shows an ambit with a more complex remoteness map, demonstrating that there are vast possibilities for the shape of an ambit if one allows non-linear ambits. When the ambits form regions in an index, one can not ensure that the search will be correct with an arbitrary f as the remoteness map. In this report, we will limit f to be linear, so that we can use the linear ambit overlap check introduced by Hetland [17] when traversing the indices. The linear ambit overlap check is described in Definition 3.

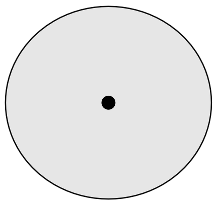


Figure 2.3: Ambit with one focus and remoteness map x_1.

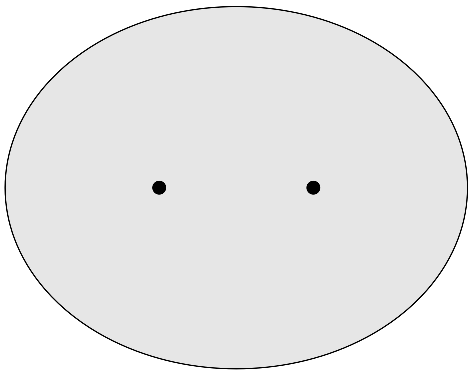


Figure 2.4: Ambit with two foci and remoteness map $x_1 + x_2$.

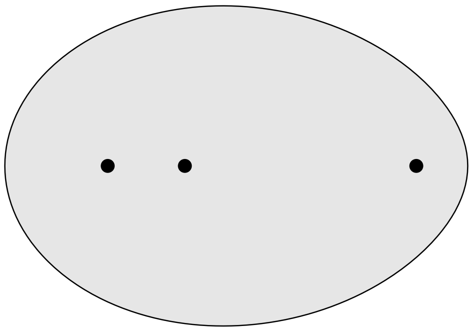


Figure 2.5: Ambit with three foci and remoteness map $x_1 + x_2 + x_3$.

Definition 3 *Linear Ambit Overlap Check [17]. Let $R := B[p, r; a]$ and $Q := B[q, s; c]$ be linear ambits for a quasimetric δ, with either a or c non-negative and $\|a\|_1, \|c\|_1 = 1$. If R and Q intersect, then*

$$r + s \geq aZc^t, \tag{2.2}$$

where z_{ij} is the distance $\delta(p_i, q_j)$ between focus p_i of R and focus q_j of Q.

2.2.5 The Goal of Metric Indexing

Measuring the distance in a metric space can be an expensive operation [29]. This is especially true in spaces with complex distance functions, such as quadratic form distance and string edit distance, and in spaces with high dimensionality [2]. Therefore, metric indexing aims to minimize the number of distances that have to be calculated. The upper bound is a linear scan, where the distance

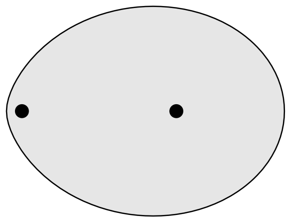


Figure 2.6: Ambit with two foci and remoteness map $x_1 + 4x_2$.

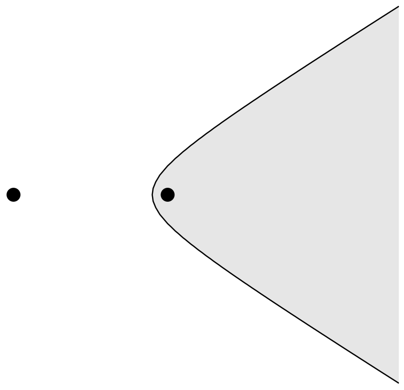


Figure 2.7: Ambit with two foci and remoteness map $x_1 - x_2$.

from the query to each data object in the data set has to be calculated.

Although reducing the number of distance calculations is the primary goal of metric indexing, there are also other essential features to the indices. One needs to be able to store the indices efficiently in secondary memory, and one can, therefore, take into account the number of I/O operations needed to access them [5]. In this report, we will not take this into account, but it can be relevant, especially when working with vector spaces with low dimensionality, as the distance functions in such spaces often are simple [7].

Other features one can consider when making indices are whether they work with discrete or continuous distance functions and whether they allow one to dynamically insert new objects after initially building the index [5].

2.2.6 The Curse of Dimensionality

Metric indexing methods are designed to structure the data so that one can discard objects that are not of interest, without having to review the object itself. The discarding of objects is based on the distance from an object to the

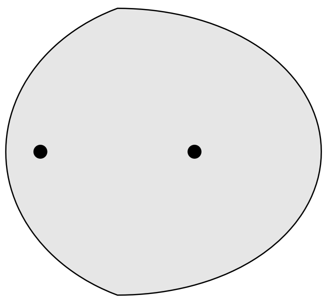
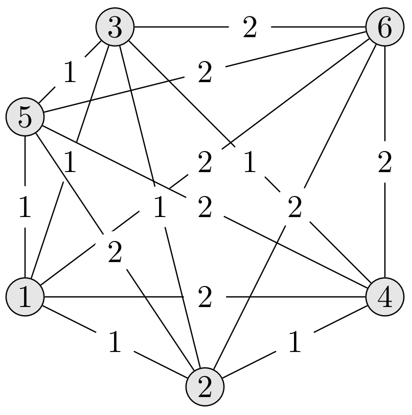


Figure 2.8: Ambit with two foci and remoteness map $f(x) = \begin{cases} x_1x_2 \text{ if } x_1 > x_2, \\ x_2^2 \text{ otherwise.} \end{cases}$

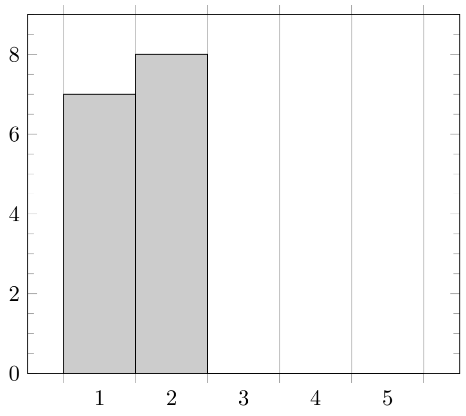
query. However, for this to be possible, one would need to be able to differentiate the distances between the objects. For example, if all distances in a data set are one, the only information obtained by comparison is whether the object is the query or not. In such a case, it would make little sense to apply a similarity search as all objects are equally similar. It would be impossible to avoid a linear scan, meaning that a traditional database scheme would work just as well [7].

The more equal the distances between the objects in a data set are, the harder it is to create an efficient index. One can create a histogram of the distances between the objects in a metric space. If this histogram is high and narrow, meaning that the mean is high and the variance is low, the distances in the data set are similar. If the mean is low, and the variance is high, meaning that the histogram is low and wide, the values of distances in the data set are spread in a larger range. The variance and mean of the histogram is referred to as the *intrinsic dimensionality* of a metric space [7]. High intrinsic dimensionality is characterized by a high mean and low variance, and low intrinsic dimensionality is characterized by a low mean and high variance. Figures 2.9 and 2.10 show two metric spaces and their corresponding distance histograms. From the histograms, we can see that the metric space in Figure 2.9 has a higher intrinsic dimensionality than the metric space in Figure 2.10. The higher the intrinsic dimensionality of a metric space is, the harder it is to make an efficient index. This phenomenon is referred to as the curse of dimensionality [7].

Intrinsic dimensionality is not the same as representational dimensionality. Nevertheless, the higher the representational dimensionality of a vectors space is, the more equidistant the data can appear. Finding efficient solutions for searching high dimensional metric spaces is one of the most important open problems in metric indexing [6].



(a) A metric space M_1.



(b) Histogram of the distances between the objects in M_1.

Figure 2.9: Example of a metric space with a relatively tall and narrow distance histogram, indicating that the intrinsic dimensionality of the space is high.

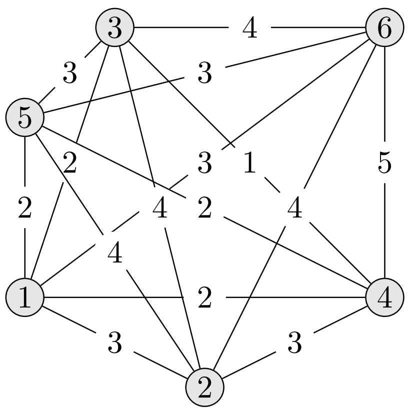
2.3 SSS-Tree

The SSS-tree [5] is a clustering-based metric indexing method that uses the so-called Sparse Spatial Selection method to choose cluster centers. This selection method makes the number of clusters in each node of the tree adapted to the complexity of the subspace it represents. On several different metric spaces, the SSS-tree has outperformed other existing clustering-based metric indexing methods [5].

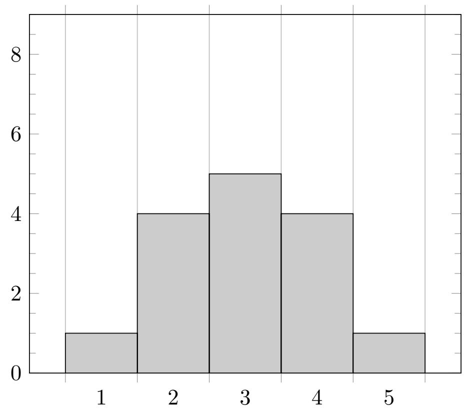
2.3.1 Sparse Spatial Selection

Sparse Spatial Selection (SSS) is a method of dynamically selecting a set of pivots, well distributed in space. This method is based on the hypothesis that if objects in a metric space are spatially sparse, they will enable us to discard more objects when they are used in search operations [5].

In SSS, when an object is further away from any existing pivot than $M\alpha$, it is selected as a new pivot [5].



(a) A metric space M_2.



(b) Histogram of the distances between the objects in M_2.

Figure 2.10: Example of a metric space with a relatively low and wide distance histogram, indicating that the intrinsic dimensionality of the space is low.

M is the maximum distance between two objects in a data set. α is a constant parameter. Experimental results have shown that using α with values in the range from 0.35 to 0.4 gives the best results [5].

SSS was initially designed for pivot selection but is used to select cluster centers for the SSS-Tree [5]. The algorithm for using SSS for clustering is described in Algorithm 1.

Algorithm 1 Algorithm for clustering using SSS.

(i) Start with a bucket containing all objects in the data set.

(ii) Calculate $M\alpha$ for the objects in the bucket.

(iii) Select a random object from the bucket as the first cluster center.

(iv) For each object in the data set, place it in the bucket of its closest cluster center if the distance from the object to its closest cluster center is less than $M\alpha$. Select it as a new cluster center otherwise.

2.3.2 Construction of an SSS-Tree

When building an SSS-tree, a Sparse Spatial Selection is performed to find the cluster centers, foci, at each level in the tree. A random object is chosen as a cluster center, and all objects that are closer to this object than $M\alpha$ will be assigned to its cluster. If an object is further away from all cluster centers than $M\alpha$, it will be selected as a new cluster center. This procedure is repeated recursively for all the clusters created until all new clusters contain fewer elements than a threshold value, δ. This process of building an SSS-tree is described in Algorithm 2.

Algorithm 2 Algorithm for building SSS-Tree.

(i) Start with a bucket containing all objects in the data set.

(ii) Calculate $M\alpha$ for the current objects in the bucket.

(iii) Select a random object from the bucket as the first cluster center.

(iv) For each object in the bucket, place it in the bucket of its closest cluster center if the distance from the object to its closest cluster center is smaller than $M\alpha$. Select it as a new cluster center otherwise.

(v) Repeat step *(ii)* to *(iv)* for all new buckets recursively until all new clusters contain fewer objects than δ. Each cluster center will correspond to a node in the SSS-tree, which will have the objects in its cluster in its sub-tree. The objects selected as cluster centers in the next recursion level will be its children. The node will have a region with the cluster center as the focus, and the distance to the object in its sub-tree that is furthest away from its cluster center as the radius.

All experiments in this report include results from queries applied to SSS-trees. We used the implementation described in Algorithm 3 when building the SSS-trees used in these experiments. The pseudo-code is simplified not to include the implementation of the sub-routines.

In Algorithm 3, the procedure *calculateMA* is used to calculate the maximum distance between two objects in a bucket. This can be done heuristically, but we chose to calculate the exact value for the experiments in this report.

Algorithm 3 Implementation of Algorithm 2.

Input: *root* is a node object which has all the other node objects in the data set as children.
Output: A node object in the form of an SSS-tree.

```
BUILDSSSTREE($root$)
 1  $root.radius$ = MAXIMUMDISTANCE($root, root.children$)
 2  if $size(root.children) \leq \delta$
 3      return $root$
 4  else
 5      $M\alpha$ = CALCULATEMA($root.children$)
 6      $newCenters = [root.children[1]]$
 7      for $child$ in $root.children$
 8          for $center$ in $newCenters$
 9              if DISTANCE($center, child$) $\leq M\alpha$
10                  APPENDTOCLOSESTCENTER($child, newCenters$)
11              else APPEND($newCenters, child$)
12  DELETEALL($root.children$)
13  for $center$ in $newCenters$
14      APPEND($root.children$, BUILDSSSTREE($center$))
15  return $root$
```

2.3.3 Example of an SSS-tree

Figure 2.11 shows the hierarchical SSS-clustering of a set of objects in the Euclidean plane, with $\delta = 1$, meaning that each leaf node will be its own cluster centers. In the first iteration of the clustering, A and H are chosen as cluster centers. In the next iteration in H's cluster, there are only two objects, meaning that none of them can be closer to each other than $M\alpha$, and both objects are therefore chosen as cluster centers. In the next iteration in A's cluster, E and D are closer to C than $M\alpha$, meaning that both of them will be contained in C's cluster. B, C, G, and F are all chosen as cluster centers. In the next iteration, C's cluster will be the only cluster containing more than δ objects, and the recursion will, therefore, stop in all other clusters. C contains two objects, and both will be chosen as cluster centers. The recursion will stop after this.

This clustering will result in an index structure shown in Figure 2.12. There are two nodes at the first level of the tree, A and H, dividing the tree into two sub-trees: sub-tree-A containing $\{A, B, C, D, E, F, G\}$ and sub-tree-H containing $\{H, I, J\}$. In the second level of sub-tree-A, there are three leaf nodes, F, B and G, and one sub-tree, sub-tree-C, containing $\{C, D, E\}$. In the second level of sub-tree-H, there are two leaf nodes I and J. As δ is set to one, one can view the leaf nodes as clusters with the center as the only element. The regions for each non-leaf node in Figure 2.12, are its corresponding ball-shaped region from Figure 2.11. The description of each region are found in Table 2.1. Even though the leaf nodes in this tree technically are cluster centers, one can view them as objects, as their corresponding regions would have a radius of 0.

Table 2.1: Description of the regions in Figure 2.11.

Cluster	Foci	Responsibilities	Remoteness Map	Radius
A	A	B, C, D, E, F, G	x_A	3.16
H	H	I, J	x_H	2.24
C	C	D,E	x_C	1.41

2.3.4 Searching

The regions in an SSS-tree are ball-shaped. Ball-shaped regions are ambits with only one focus, where the remoteness map x_1 is the distance to the focus.

When searching for a query Q in an SSS-tree, an overlap check is performed to check whether the query overlaps with the region of the current node or not. An overlap indicates that the query might be found in the sub-tree of the current node and that one should continue to examine this sub-tree further. The sub-trees of the nodes whose regions do not overlap with the query are discarded.

This traversal process is described in Algorithm 4. It is a modified version of Hetland's Sprawl Traversal Algorithm [17]. In step (*iv*), Definition 3 is used to check for overlap.

All the weighted indices that will be presented in the following chapters are modified versions of the SSS-tree. This means that when applying queries to these indices, a modified version of Algorithm 4 is used.

Algorithm 4 Algorithm for traversing an SSS-Tree.

Input: An SSS-tree T, a distance function $d(x, y)$ and a query Q with focus f_q and radius R_q.
Output: A list containing all objects from T that match Q.

(i) Start with a queue Qu with T as its only element, and an empty collection of results, Re.

(ii) Dequeue Qu, and set the current tree, T_c, to be this element.

(iii) Check if the root r_c of T_c is inside Q by checking if $d(r_c, f_q)$ is smaller than or equal to R_q. If this is the case, place r_c in Re.

(iv) Check if the region of node r_c with radius R_c overlaps with Q. This is done by checking whether $d(r_c, f_q)$ is smaller or equal to $R_q + R_c$. If this is the case, add all the children of r_c to Qu.

(v) Repeat step (ii) to (v) for the next element in Qu. When Qu is empty, return Re.

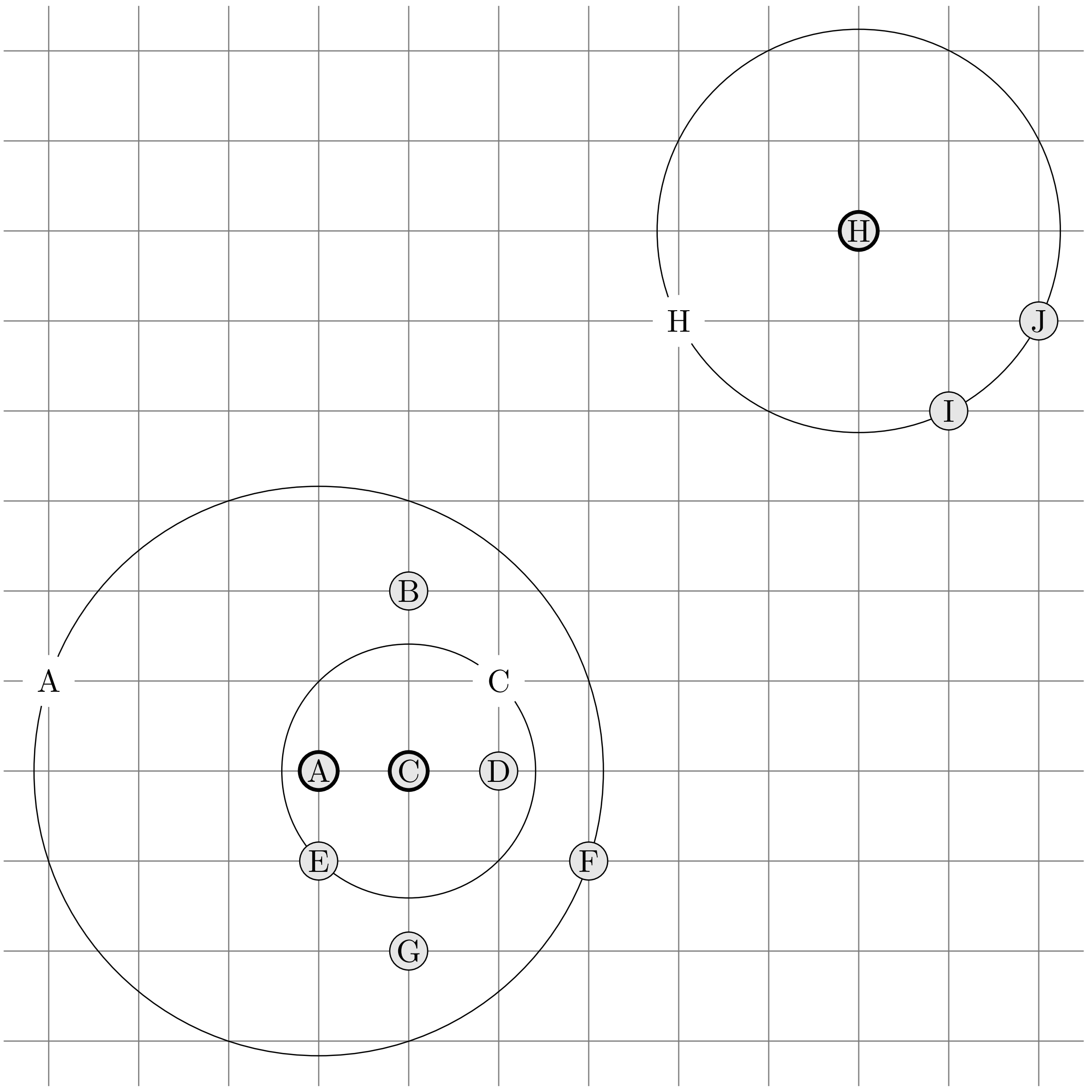


Figure 2.11: The circles represent the recursive SSS-clustering a set of objects in the Euclidean plane and their corresponding regions in an SSS-tree.

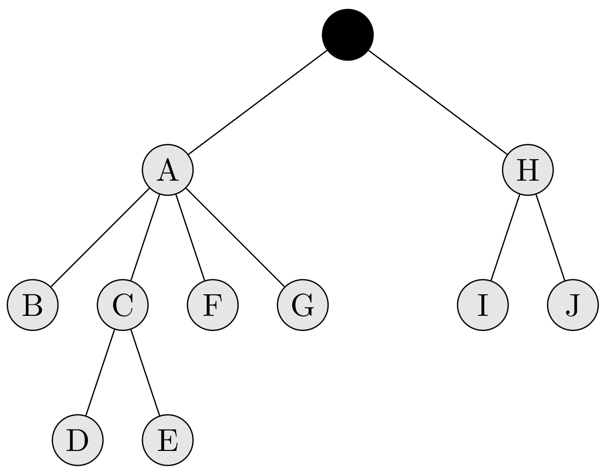


Figure 2.12: An SSS-tree built from the clustering in Figure 2.11.

Chapter 3

The WSSS-tree, CSSS-tree, and Hypotheses 1 and 2

In this chapter, we present the following metric index structures: the WSSS-tree and the CSSS-tree. Both of these are adaptations of the SSS-tree where all sibling nodes use each other as foci in their regions. We will explain the motivation behind and describe how to build and search through these indices. Furthermore, we will present the two first hypotheses tested in this report: hypotheses 1 and 2.

3.1 WSSS-tree

When searching in an SSS-tree, the overlap check should give as few false-positive overlaps as possible. Allowing multiple foci to describe each region will give more possibilities of shapes the regions can hold, and could, therefore, result in fewer false-positive overlaps [17]. The foci of a region can be weighted differently, which will affect the shape of the region. These regions can be trained to minimize overlap with a set of training queries, while still covering all of the same data points as the corresponding ball region in an SSS-tree.

3.1.1 Motivation

When traversing an SSS-tree, what resembles a breadth-first search is performed. This means that all sibling nodes in a tree need to be examined before

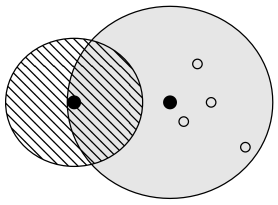


Figure 3.1: The grey area represents a ball shaped uni-focal region in the Euclidean plane in an SSS-tree, containing four objects in addition to its focus. The hatched circle represents a range query.

one can move on to their children, and the distance from the query to each sibling node needs to be calculated.

The goal of creating a metric index is to minimize the number of times one has to calculate the distance between two objects. As one always has to examine all the sibling nodes at one level in a sub-tree, adding all sibling nodes as each other's foci will not increase the number of times the distance is calculated.

Having multiple foci can give fewer false-positive overlaps, and adding all sibling nodes as foci will not increase the number of times the distance is calculated. Thus, it is reasonable to assume that doing this might improve the performance of an SSS-tree.

Even when using all sibling nodes as foci, the regions could be equivalent to the regions in an SSS-tree if one sets the weight of the original focus to one, and weights of the rest of the foci to zero. This means that if the optimal region to avoid a set of training queries is the original SSS-region, such a region will be chosen.

We will call a tree with all sibling nodes as foci weighted to avoid a set of training queries a WSSS-tree (Weighted SSS-tree).

Figure 3.1 shows a region in an SSS-tree that overlaps with a query, even though none of its objects are within the query. In Figure 3.2, the region is from a WSSS-tree and uses a sibling node as a focus in addition to itself. This region contains all the same objects but avoids overlapping the query.

3.1.2 Tree construction

Our implementation for constructing a WSSS-tree is described in Algorithm 5, and is a modified version of Algorithm 3.

Algorithm 5 differs from Algorithm 3, in the way that each region's foci are selected, as well as in the way it assigns each region's weights and radius.

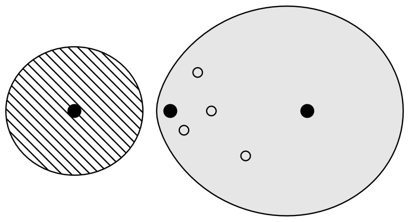


Figure 3.2: The grey area represents a multi-focal region in the Euclidean plane in an WSSS-tree, containing the same objects as the region in Figure 3.1. The hatched circle represents the same range query as in Figure 3.1.

Algorithm 5 Algorithm for constructing a WSSS-tree.

Input: *root* is a node object which has all the other node objects in the data set as children, and *trainingQueries* is a set of queries.
Output: A node object in the form of an WSSS-tree.

```
BuildWSSSTree($root, trainingQueries$)
 1  assignOptimzedWeigthsAndRadius($root, trainingQueries$)
 2  if $size(root.children) \leq \delta$
 3      return $root$
 4  else
 5      $M\alpha$ = calculateMA($root.children$)
 6      $newCenters = [root.children[1]]$
 7      for $child$ in $root.children$
 8          for $center$ in $newCenters$
 9              if distance($center, child$) $\leq M\alpha$
10                  appendToClosestCenter($child$)
11              else append($newCenters, child$)
12  deleteAll($root.children$)
13  for $center$ in $newCenters$
14      $center.foci = newCenters$
15      append($root.children$, BuildWSSSTree($center, trainingQueries$))
16  return $root$
```

For each level in a sub-tree, the foci of all regions at this level are set to the cluster centers found using SSS, *newCenters*. The radius and weights of each region are found using *assignOptimzedWeights*, which calculates and assigns the optimized weights through the linear program, created by Hetland, described below [17]:

In the following, let m be the number of foci in the region, and n the number of elements the region is responsible for covering. $\mathbb{1} = [1...1]^t$ is a column vector of size m containing only ones. $X := x_{ij}$ is a non-negative $m \times n$ matrix. The matrix shows the distance from each focus to the n points the region is responsible for covering. $\hat{z} = E[z]$ is a column vector of size m containing the average distance from a focus to a set of training queries. $a = u - v$ is a row vector containing the weights of the foci. The vector is split into u and v to make sure that the result's absolute value is calculated.

$$\begin{array}{lll} \text{maximize}_{u,v,r} & u\hat{z} - v\hat{z} - r & \\ \text{subject to} & u\mathbb{1} + v\mathbb{1} & = 1, \\ & uX - vX - r & \leq 0, \\ \& & u, v & \geq 0. \end{array} \tag{3.1}$$

Linear Program 3.1 uses the average distance to the training queries from each focus to optimize the weights of the foci. This means that the regions will be created to try to avoid overlapping with the average of the training queries. We want the regions to avoid overlapping with as many of the queries the index will be applied to as possible. Therefore, the average feature vector of the training queries z needs to represent the queries that the index will be applied to.

3.1.3 Example of a WSSS-tree

When creating a WSSS-tree using the same data objects from the Euclidean plane as in figs. 2.11 and 2.12, the clusters will be the same as for the SSS-tree, as the WSSS-tree also uses SSS for clustering. This means that Figure 2.12 will also describe the WSSS-tree, only with different regions attached to each cluster center, A, H, and C. These regions will now be shaped to move away from the average of a set of training queries, and are shown in Figure 3.3. The description of these regions is found in Table 3.1. The dashed regions show the original SSS-regions for the data set. The solid regions show the WSSS-regions created using all sibling nodes as foci optimized to avoid Q, the average point of a set of training queries. The original SSS-regions overlap Q, even though no

Table 3.1: Description of the regions in Figure 3.3.

Cluster	Foci	Responsibilities	Remoteness Map	Radius
A	A, H	B, C, D, E, F, G	$0.66x_A + 0.34x_H$	4.67
H	A, H	I, J	x_H	2.24
C	B, C, F, G	D,E	$0.34x_F + 0.66x_G$	1.96

objects it is responsible for overlap Q. The WSSS-regions avoid overlapping Q and allow for a more snug fit around the objects in general.

3.1.4 Searching

When searching a WSSS-tree, one can use Algorithm 4, but with a slight modification in step (*iv*), as the regions now are ambits with multiple foci, and the remoteness map is the weighted sum of distances to each focus. Step (*iv*) will now be an implementation of Definition 3. When checking if the regions of the node and the query overlap, one can use the weighted sum of the distances from the query focus to the foci of the node, $w_i \cdot d(r_i, f_q)$ for all $i \in 1...m$, instead of just the distance between the query focus and the focus of the node, $d(r_c, f_q)$, as done when traversing the SSS-tree. To check for overlap, one checks if this weighted distance is greater than $R_q + R_c$, just as for the SSS-tree.

The premise for the WSSS-tree to perform better than the SSS-tree is that one only has to calculate the distance from the query to all sibling nodes once. This means that one has to temporarily store the distance of the query to each of the sibling nodes for each level when traversing each sub-tree in a WSSS-tree.

3.1.5 Space Complexity

Even though the WSSS-tree will not cause any more distance calculations than the SSS-tree, it requires more space in memory. For each region in an SSS-tree, one only has to store the focus and the region's radius. For each region in a WSSS-tree, one also has to store one weight for each of its sibling nodes. This means that one has to store as many weights as the number of siblings to each non-leaf node squared, in addition to the information one would have to store in an SSS-tree.

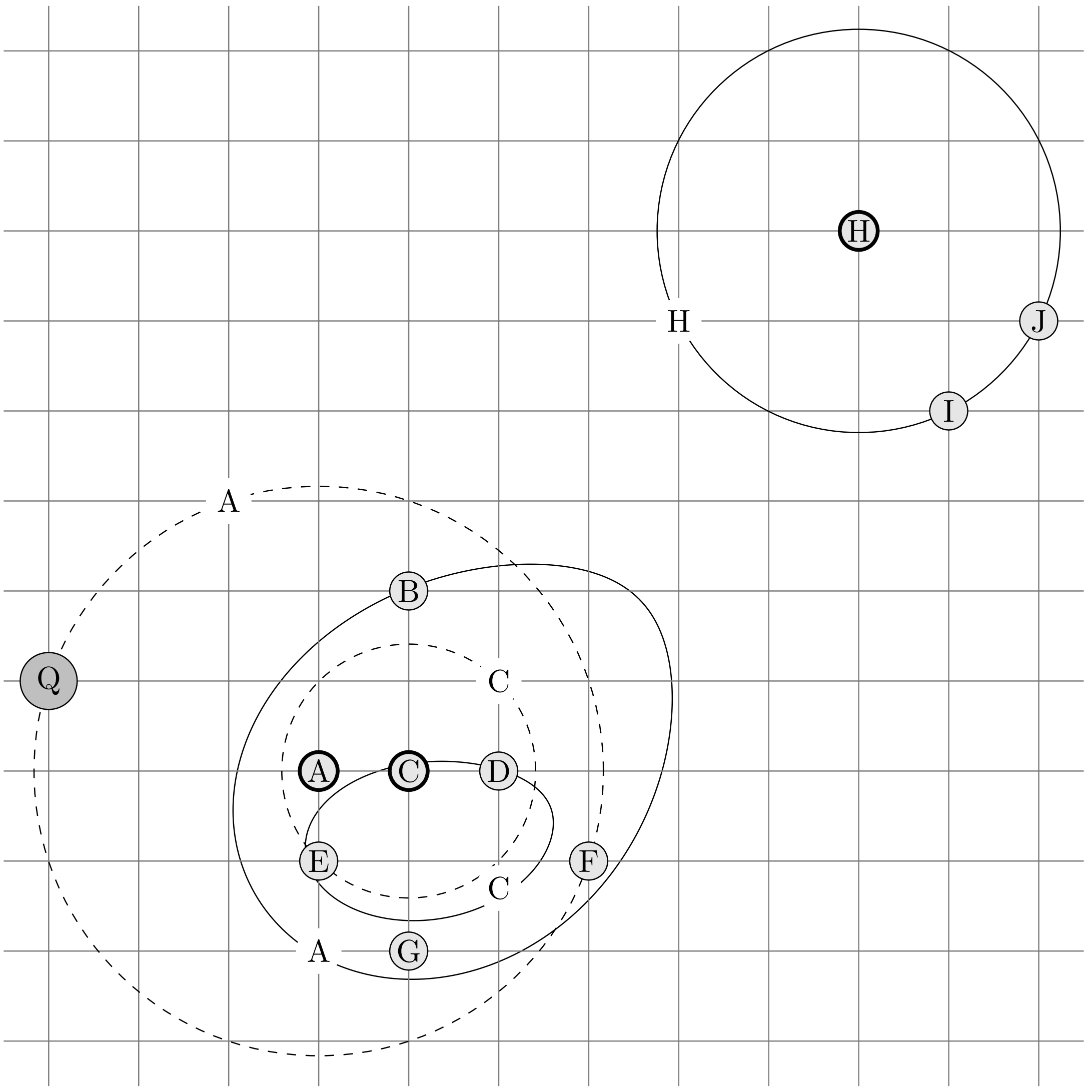


Figure 3.3: The recursive SSS-clustering set of objects in the Euclidean plane and their corresponding SSS-regions (dashed) and WSSS-regions (solid).

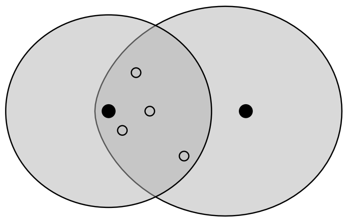


Figure 3.4: The grey area shows the two facets of a multi-focal region in the Euclidean plane in a CSSS-tree, containing the same objects as the regions in Figure 3.1 and Figure 3.2.

3.2 CSSS-tree

We will now present the CSSS-tree, which is a combination of the WSSS-tree and SSS-tree. The CSSS-tree can never perform worse than any of them, and can, therefore, be used as a lower bound for their performance.

3.2.1 Motivation

The weights of the regions in a WSSS-tree are optimized to avoid the average feature vector of a set of training queries and could be set to one and zero if this gives the highest objective value in the linear program. Nevertheless, the SSS-tree can still, in some cases, outperform the WSSS-tree, if it is, for example, applied to queries that are not well represented by this average feature vector. To ensure that an out-performance by the SSS-tree never happens, one can combine the two trees so that overlap checks are performed for both trees without causing any more distance calculations. This means that the combined tree will have two facets, the ambit of the SSS-tree and the ambit of the WSSS-tree. We will call such a tree, a CSSS-tree (Combined SSS-tree).

Figure 3.4 shows the two facets of a CSSS-tree. A query needs to overlap with the intersection of these two facets, which is dark grey, if one is to explore the node's sub-tree. This means that the CSSS-tree will never give more overlaps than the other trees, so its performance will always be at least as good as the other trees'.

3.2.2 Tree construction

When constructing a CSSS-tree, one combines the implementation of the SSS-tree, Algorithm 3, and the implementation of the WSSS-tree, Algorithm 5. The resulting implementation is described in Algorithm 6.

Algorithm 6 Algorithm for constructing a CSSS-tree.

Input: *root* is a node object which has all the other node objects in the data set as children and *trainingQueries* is a set of queries. All node objects in the data set are initialized with themselves as $sssFocus$.

Output: A node object in the form of an CSSS-tree.

```
BUILDCSSSTREE($root, trainingQueries$)
 1  ASSIGNOPTIMZEDWEIGTHSANDWSSSRADIUS($root, trainingQueries$)
 2  $root.sssRadius$ = MAXIMUMDISTANCE($root, root.children$)
 3  if $size(root.children) \leq \delta$
 4      return $root$
 5  else
 6      $M\alpha$ = CALCULATEMA($root.children$)
 7      $newCenters = [root.children[1]]$
 8      for $child$ in $root.children$
 9          for $center$ in $newCenters$
10              if DISTANCE($center, child$) $\leq M\alpha$
11                  APPENDTOCLOSESTCENTER($child$)
12              else APPEND($newCenters, child$)
13  DELETEALL($root.children$)
14  for $center$ in $newCenters$
15      $center.wsssFoci = newCenters$
16      APPEND($root.children$, BUILDCSSSTREE($center, trainingQueries$))
17  return $root$
```

In Algorithm 6, each node object has both a facet from the SSS-tree, an SSS-facet, and a facet from the WSSS-tree, a WSSS-facet. The SSS-facet consists of a focus and a radius, whereas the WSSS-facet consists of a set of foci, a radius, and weights of the foci.

All nodes are initialized with themselves as the focus of the SSS-facet. The radius of the SSS-facet is found by calculating the maximum distance from themselves to a child, using *maximumDistance*.

The foci of the WSSS-facet are found the same way as in Algorithm 5. The radius and weights are found using $assignOptimzedWeigthsAndWsssRadius$, which uses the linear program in Formula 3.1 to optimize the weights.

Table 3.2: Description of the regions in Figure 3.5.

Cluster	Facet	Foci	Responsibilities	Remoteness Map	Radius
A	SSS	A	B, C, D, E, F, G	x_A	3.16
A	WSSS	A, H	B, C, D, E, F, G	$0.66x_A + 0.34x_H$	4.67
H	SSS	H	I, J	x_H	2.24
H	WSSS	A, H	I, J	x_H	2.24
C	SSS	C	D,E	x_C	1.41
C	WSSS	B, C, F, G	D,E	$0.34x_F + 0.66x_G$	1.96

3.2.3 Example of a CSSS-tree

The CSSS-tree built using the same example data set from Figure 2.11 will be the same as the tree in Figure 2.12, and will have both the SSS-regions and WSSS-regions from Figure 3.3 as facets. The description of the facets is found in Table 3.2. The intersection of the two facets will be each cluster center's region in the CSSS-tree. The CSSS-regions are shown in Figure 3.5. The dotted area is A's region, the hatched area is H's region, and the light grey area is C's.

3.2.4 Searching

When searching a CSSS-tree, Algorithm 4 can again be used, but in step (*iv*) one uses the region overlap check for both the SSS-tree and WSSS-tree's ambits using *sssRadius* with *sssFocus* and *wsssRadius* with *wsssFoci* respectively. When traversing a tree with two facets, the traversal algorithm requires both of these checks to give a positive result for the region to be looked into further. The distance is still only calculated once for each node on each level of each sub-tree, meaning that the cost of searching a CSSS-tree is equal to the cost of searching its corresponding SSS-tree.

3.2.5 Space Complexity

The CSSS-tree needs to save information about both the SSS-facet and the WSSS-facet for each region. This means that the CSSS-tree will use the amount of space that the WSSS-tree would, with the addition of the radius of the SSS-tree facet for each region.

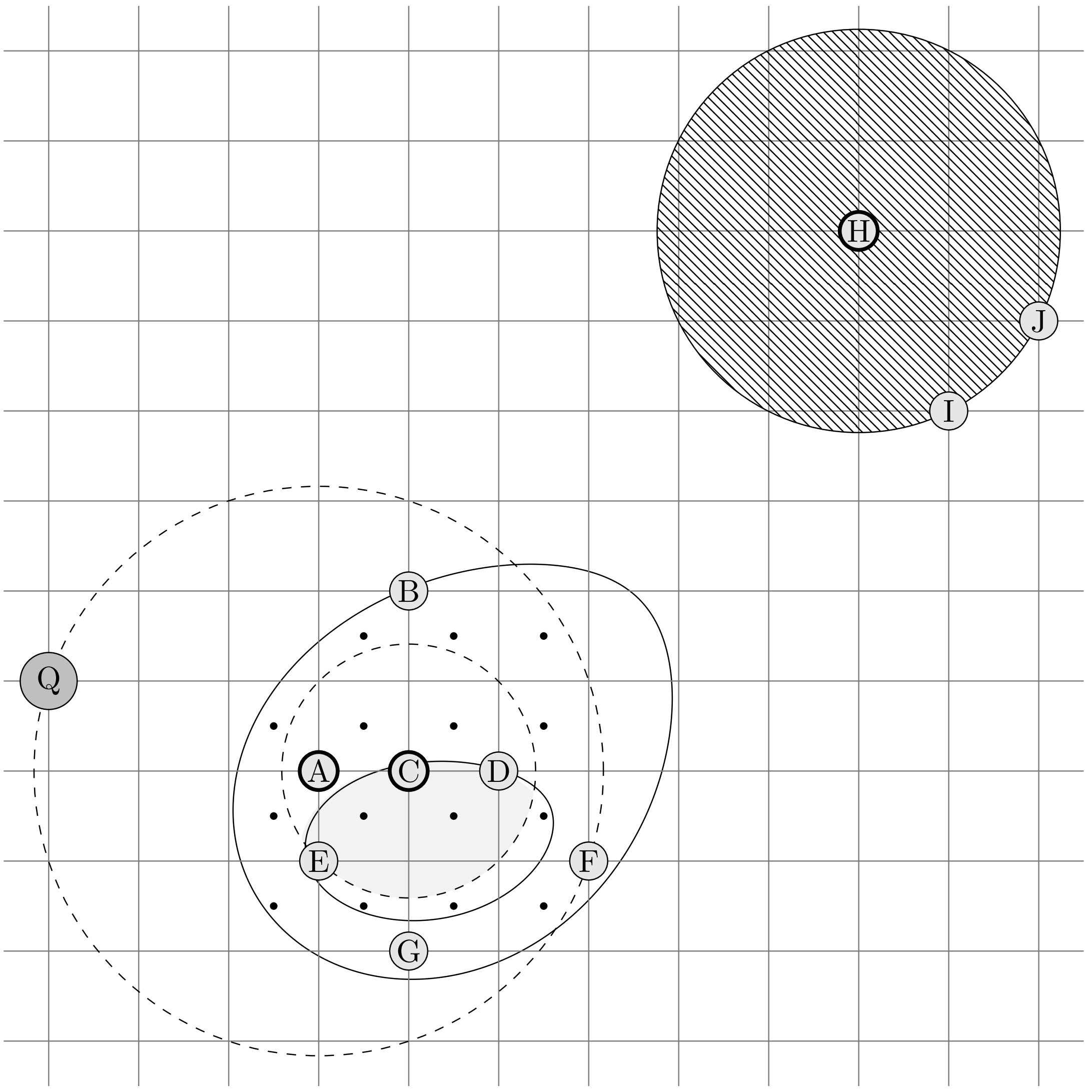


Figure 3.5: The recursive SSS-clustering a set of objects in the Euclidean plane and their corresponding CSSS-regions.

3.3 Hypotheses

One goal in this report is to find scenarios where the weighted trees will outperform the SSS-tree. To find such scenarios, one has to consider the selection of training queries. Using queries from one cluster form the basis for Hypothesis 1 and using queries from multiple clusters form the basis for Hypothesis 2. We will now discuss the motivation behind choosing these distributions.

3.3.1 Selection of Training Queries

The weights of each regions' foci in a WSSS-tree are trained to minimize the chance of overlap with a set of training queries, by maximizing the average distance from the foci to these. Therefore, one has to choose the set of training queries deliberately. If the training queries are chosen from a different distribution than the queries that the index structure will be used on, they can decrease the index structure's performance, as the regions will be misguidedly directed.

One can think that choosing training queries uniformly from the data set will be a solution if one does not know the distribution of the queries that the index will be used on. Instinctively, one would think that this would mean that the regions will try to avoid queries from all over the data set, and through this, make the regions fit more snugly around the objects it is responsible for. However, the linear program used to optimize the weights only uses the average distance of each focus to the training queries. This means that the weights will, in practice, try to avoid this one query object only, which is the average of all the training queries. Therefore, if one is to choose training queries uniformly from the data set, the average query of these will be close to the true average of the data set. This point would not make sense to avoid when trying to make a more efficient index structure if the queries can come from anywhere in the metric space.

3.3.2 Hypothesis 1

If one expects the queries to be from one specific area in the space of the data set, trying to avoid the point that describes the average distance from the queries in this cluster to the foci can make sense and cause an improvement of performance. In this case, the regions will be directed away from this average point, resulting in the regions to be directed away from the cluster that the queries will come from. This is illustrated in Figure 3.2. Therefore, in such a situation, one can expect the WSSS-tree to outperform the SSS-tree.

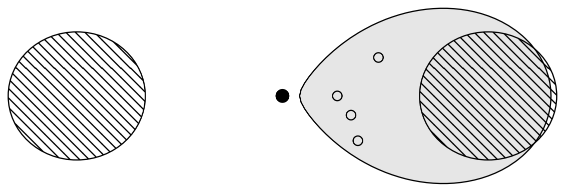


Figure 3.6: The grey area represents a region constructed to cover the objects it is responsible for, while avoiding the average of the training queries, represented by the black point. The hatched circles represent the clusters the training queries come from.

From this, we formed Hypothesis 1: *The WSSS-tree will outperform the SSS-tree if the queries come from one cluster.*

3.3.3 Hypothesis 2

If the distribution of the training queries resembles more than one cluster, and these are located at opposite sides of the space of the data set, one again encounters the problem that the average distance between these and the foci is not representative of any of the clusters that the actual queries will come from. This is illustrated in Figure 3.6. Hence, in such a situation, one can not expect the WSSS-tree to outperform the SSS-tree to the same degree as when the queries come from one cluster.

From this, we formed Hypothesis 2: *The WSSS-tree will lose a part of its advantage over the SSS-tree when the queries come from more than one cluster if the clusters are so far apart that they would not be classified as one larger cluster.*

Chapter 4

The W2SSS-tree and Hypothesis 3

In this chapter, we present the W2SSS-tree, a modified version of the WSSS-tree designed to perform better on queries from two clusters. Additionally, we present the third and final hypothesis that is tested in this report: Hypothesis 3.

4.1 The W2SSS-tree

The W2SSS-tree is a modified version WSSS-tree, with two facets instead of one for each region. Each of these facets is trained to avoid one cluster of queries each.

4.1.1 Motivation

To modify the WSSS-tree to work better for queries from multiple clusters, one could implement multiple facets, as described by Hetland [17].

The opposite extreme of using only one facet would be to allow an arbitrary number of facets. This would allow the defining polyhedron to be the inclusion-wise minimum, a convex hull of the responsibility vectors, and be the optimal coefficients for the region [17]. However, this would make the number of facets exponential of the number of responsibilities. This would not cause more distance calculations but would increase the use of memory. Even though we have only used distance calculations as a measure of cost, one can not neglect memory usage for index structures. The point of clustering-based index structures is that they use less space than methods that require the entire distance matrix to be loaded in memory, such as AESA [25], which was considered the baseline for similarity searches for more than 20 years [12]. Such methods would be faster than an index structure with an arbitrary number of facets, meaning that such an index structure would be inefficient in both space and time.

Between the two extremes of having an arbitrary number of facets, and just a single one, one could have as many facets as there are clusters in the training queries. This would mean that in Experiment 2, where we use training queries from two different clusters, one would have two facets. For this to be beneficial, one can optimize the facets to avoid one cluster of training queries each, instead of optimizing one facet to avoid the average of the two clusters.

The number of facets and number of clusters, k, can either be pre-decided or found through clustering of the training queries. If the number of clusters is fixed, one runs a clustering algorithm on the training queries to cluster them into the decided number of clusters. Either way, the clustering of the queries will decide which facets are responsible for which training queries [17].

We will call a WSSS-tree with two facets, trained on two different clusters of queries, a W2SSS-tree (Weighted 2-Facet SSS-tree).

4.1.2 Tree construction

Constructing a W2SSS-tree is done in the same way as constructing a WSSS-tree, except that the weights are optimized twice. The resulting implementation is described in Algorithm 7.

In Algorithm 7, each node object has two facets, $facet1$ and $facet2$. The foci of both facets will be the same, and therefore only set once, the same way as in Algorithm 5. The radius and weights of both facets are found using $optimzeWeigthsAndRadius$, but with different sets of training queries as input: $trainingQueries1$ and $trainingQueries2$. For the W2SSS-tree to work as intended, these two sets of training queries should be representative of one cluster of queries each.

4.1.3 Example of a W2SSS-tree

The W2SSS-tree created from the same data set as in Figure 2.11, will give the same tree structure as in Figure 2.12, as it also uses SSS to cluster the data. The regions of the tree are trained to avoid two clusters of training queries,

Algorithm 7 Algorithm for constructing a W2SSS-tree.

Input: *root* is a node object which has all the other node objects in the data set as children and two sets of training queries: $trainingQueries1$ and $trainingQueries2$.

Output: A node object in the form of an W2SSS-tree.

```
BuildW2SSSTree($root, trainingQueries1, trainingQueries2$)
 1  $root.facet1$ = optimzeWeigthsAndRadius($root, trainingQueries1$)
 2  $root.facet2$ = optimzeWeigthsAndRadius($root, trainingQueries2$)
 3  if $size(root.children) \leq \delta$
 4      return $root$
 5  else
 6      $M\alpha$ = calculateMA($root.children$)
 7      $newCenters = [root.children[1]]$
 8      for $child$ in $root.children$
 9          for $center$ in $newCenters$
10              if distance($center, child$) $\leq M\alpha$
11                  appendToClosestCenter($child$)
12              else append($newCenters, child$)
13  deleteAll($root.children$)
14  for $center$ in $newCenters$
15      $center.foci = newCenters$
16      append($root.children$, BuildW2SSSTree($center, trainingQueries1$,
17      $trainingQueries2$))
18  return $root$
```

Table 4.1: Description of the regions in Figure 4.1.

Cluster	Facet	Foci	Responsibilities	Remoteness Map	Radius
A	Q_1	A, H	B, C, D, E, F, G	$0.66x_A + 0.34x_H$	4.67
A	Q_2	A, H	B, C, D, E, F, G	x_A	3.16
H	Q_1	A, H	I, J	x_H	2.24
H	Q_2	A, H	I, J	x_A	9.43
C	Q_1	B, C, F, G	D,E	$0.34x_F + 0.66x_G$	1.96
C	Q_2	B, C, F, G	D,E	x_G	2.24

represented by their average points Q_1 and Q_2. The resulting regions are shown in Figure 4.1, and the description of the regions are found in Table 4.1. One facet is optimized to avoid Q_1, illustrated with solid lines. The other facet is optimized to avoid Q_2, illustrated with dashed lines. The intersection of the two facets will be each cluster center's region. The dotted area is A's region, the hatched area is H's region, and the light grey area is C's region.

4.1.4 Searching

When searching a W2SSS-tree, Algorithm 4 can again be used, but in step (*iv*), one uses the region overlap check for both facets. When traversing a tree with two facets, the traversal algorithm requires both of these checks to give a positive result for the region to be looked into further. The distance is still only calculated once for each node on each level of the tree, meaning that the cost of searching a W2SSS-tree is equal to the cost of searching its corresponding SSS-tree.

4.1.5 Space Complexity

The W2SSS-tree will need to save twice as many weights and radii for each non-leaf node than its corresponding WSSS-tree.

4.2 Hypothesis 3

We believe that a W2SSS-tree will perform better than a WSSS-tree if the queries come from two clusters. This is because we construct two facets for each region that avoid one cluster of training queries each, instead of constructing

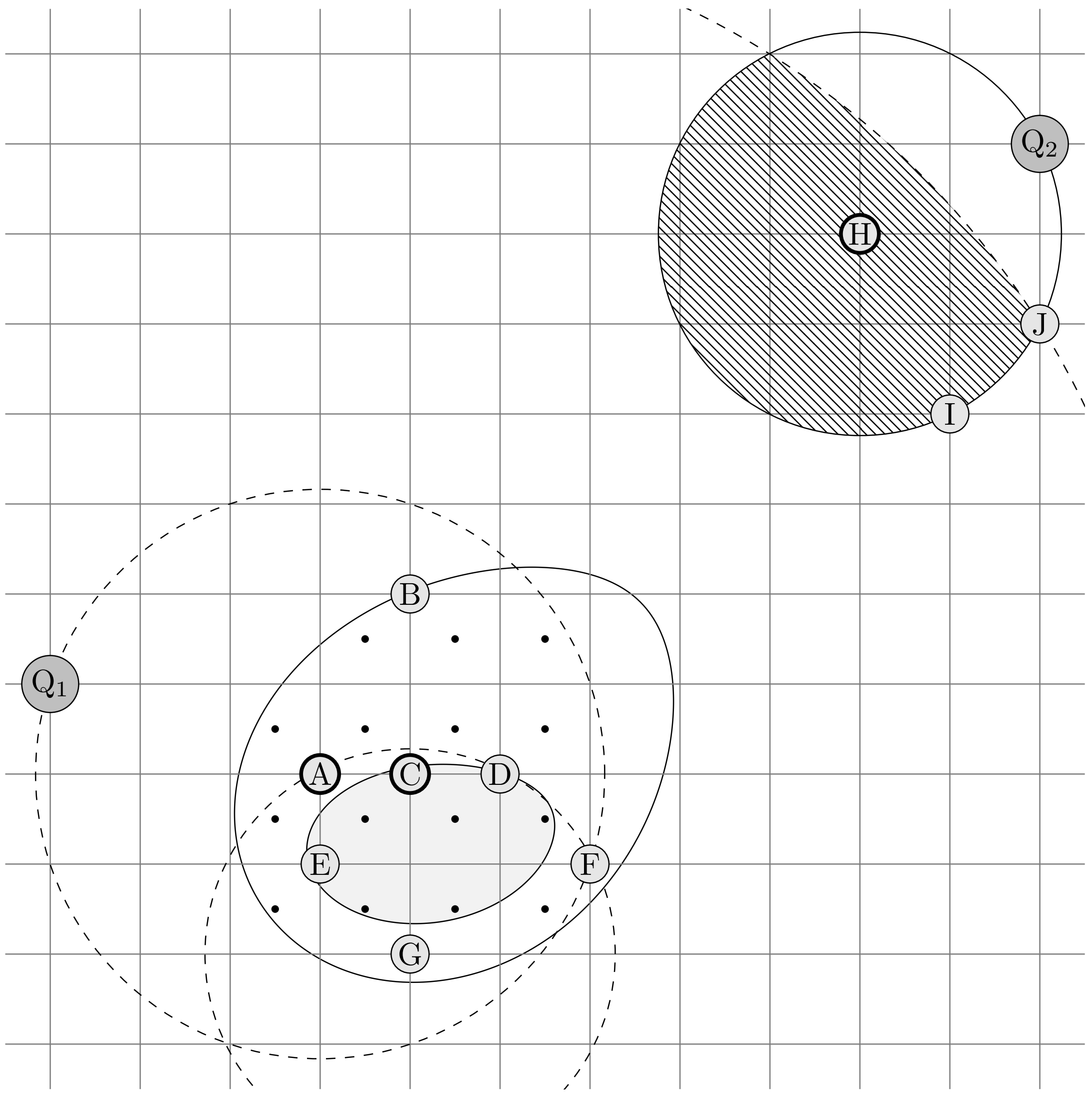


Figure 4.1: The recursive SSS-clustering of a set of objects in the Euclidean plane and their corresponding W2SSS-regions.

one region to avoid a single point that is not necessarily near any of the actual training queries. This means that the W2SSS-tree will try to avoid the two actual query clusters instead of none of them, which should result in better performance, and is illustrated in figs. 3.6 and 4.2.

Figure 3.6 illustrates a region that is constructed to avoid the average point of training queries from two different clusters, while still covering the objects it is responsible for. This region intersects with almost all queries from one of the two training query clusters, contradicting the purpose of using training queries.

Figure 4.2 illustrates a region consisting of two different facets, each one constructed to avoid the average of one of the two training query clusters. The intersection of these does not overlap with any of the training query clusters, making this region more suited to avoid the training queries, than the region in Figure 3.6.

From this we formed the following hypothesis, Hypothesis 3: *The W2SSS-tree will outperform both the SSS-tree and the WSSS-tree when the queries come from two clusters if the clusters are so far apart that they would not be classified as one larger cluster.*

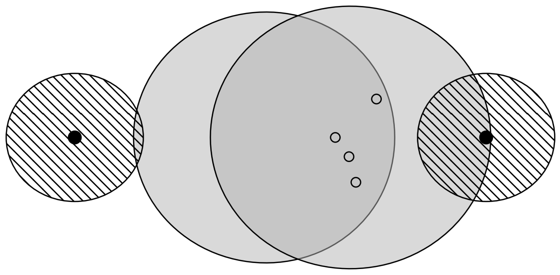


Figure 4.2: The grey area represents two facets, both constructed to cover the objects the region is responsible for while avoiding the query average from one of the two training query clusters each, represented by the black points. The hatched circles represent the clusters the training queries come from.

Chapter 5

About the Experiments

This chapter will present the set-up of the three experiments performed in this report, as well as provide information about the data sets and details about our implementation of the indices.

5.1 Data sets

To ensure that the results of the experiments can translate to a range of use cases, we used multiple data sets in the simulations.

As the use case for indexing is usually to perform searches on real-life data, we ran our experiments on the following real-life data sets: The Copenhagen Chromosome data set and the Corel Image Collection. We used two real-life data sets to demonstrate the trees' behavior in more than one metric space. These two different real-life data sets cover the use cases of string edit distances and color moments in images. We chose string edit distances and images, as it has been previously shown that the SSS-tree outperforms other existing indexing methods on these types of data sets [5].

Nevertheless, generating synthetic data sets allows one to control the properties of the data, such as size and dimension.

The SSS-tree has been shown to perform better than other existing indexing methods on generated data sets of dimension 10. Therefore, we tested the index structures on data sets of this dimension. As the dimension of the space that one is to index grows, the search performance of existing indices becomes worse [3]. For this reason, it is interesting to test the WSSS-tree, CSSS-tree, and W2SSS-tree on data sets with different dimensions. Therefore, we tested our indices on generated data sets of dimension 5 and 15, in addition to 10.

Previously, the SSS-tree has been shown to perform better than other existing indexing methods on synthetic data sets of size 100 000 [5]. The generated data sets used for the experiments are, therefore, also of this size.

We ran the experiments on generated data sets of two different distributions: uniformly distributed vectors from the unitary cube and Gaussian clusters. Preliminarily, we also experimented with synthetically generated data sets with clusters from Gaussian Mixture Models, as these are more oblong [9], and could potentially benefit more from using multi-focal ambits as regions. However, these results are not included in this report, as the performance differed very little from the Gaussian clusters.

More information about, and the rationale for choosing, each data set will follow in the following subsections.

5.1.1 Copenhagen Chromosome Data Set

The Copenhagen Chromosome data set [26] consists of 4200 chromosome strings. The strings are difference coded strings with a six level representation of the profile. An example of such a string is:
AA==a==E===d==A==a=Aa=A=a=b.

We chose to use a data set of chromosome strings instead of words from a natural language, as the distance between these strings could in a bigger range because they are longer. This should result in higher variance in the distances, meaning that the intrinsic dimensionality should be lower, and the indices' performance should improve [12].

The edit distances are already calculated in a distance matrix, so the distance matrix is used for the simulation testing. Since the data set is only 4200 in size, the whole distance matrix can fit in memory.

The edit distances between the difference coded chromosome strings are calculated using Formula 5.1 and Table 5.1:

$$\begin{aligned} c_k(x \to y) &= |x - y| \\ c_k(\epsilon \to x) = c_k(x \to \epsilon) &= \begin{cases} |x| \text{ if } k < |x|, \\ k \text{ else.} \end{cases} \end{aligned} \tag{5.1}$$

We used the value of 2.5 for k in the simulations.

Table 5.1: Difference code for the Copenhagen Chromosome data set.

Difference	−5	−4	−3	−2	−1	0	1	2	3	4	5
Code	e	d	c	b	a	=	A	B	C	D	E

5.1.2 The Corel Image Collection

The Corel Image Collection [18] is a collection of 68 040 photo images from various categories.

The following features were extracted for each image: the color histogram, the color histogram layout, the co-occurrence texture, and the color moments. We used the Euclidean distance between the Color Moments of two images to find the similarity between these images.

The color moments of an image consists of 9 dimensions, with the mean, standard deviation, and skewness for each value H, S, and V in the HSV color space. The HSV color is an alternative representation of the RGB color space where the H represents hue, S represents saturation, and V represents value [27].

5.1.3 Random Vectors from the Unitary Cube

A common way to generate synthetic data sets of different dimensions is by generating random vectors uniformly distributed in the unitary cube [12].

We used three generated data sets of this type, each of size 100 000, with dimensions 5, 10, and 15.

We used the Euclidean distance between the vectors to find the similarity between each data object.

5.1.4 Gaussian Clusters

While uniformly distributed data is useful to simulate how indices behave in spaces of different dimensions, it can also be interesting to look at data that has a different distribution. Indexing can work well on data that is naturally clustered, and one can generate Gaussian clusters to create a synthetic clustered data set. Gaussian clusters are useful as a lot of real-life data closely follows a Gaussian distribution [9].

We used three generated data sets, each of size 100 000, with 10 clusters consisting of 10 000 objects each, with dimensions 5, 10, and 15. The mean of the distribution of each cluster was a vector the size of the dimension, with random values between 0 and 1. The standard deviations of the Gaussian distributions were all set to one.

We used the Euclidean distance between the vectors to find the similarity between each data object.

5.2 Experiment Setup

When building the indices using algorithms 3 and 5 to 7, the threshold value δ was set to 10, and we used the value of 0.4 as α.

In all experiments, we tested the indices by searching for range queries. These queries return all objects whose distances to the query object do not exceed a set radius [15]. When optimizing the weights of the foci when building the indices, one does not consider the radius of the training queries, and, thus, the radius of all training queries was set to zero.

The training and test queries were uniformly drawn from the same distribution. The distribution of the queries was either one or two clusters with a total size of 150. For each index that was built, we sampled 50 of these to use as training queries. We tested the index on the remaining 100 queries.

The number of queries needs to be big enough to provide accurate results, but could not be arbitrarily big for this project, as the computational resources were not unlimited. The number of queries in this experiment was chosen to be a trade-off between these concerns.

Every training and test query was selected from the data sets that the indices were built from, meaning that each search was exact, and the smallest number of objects that could be retrieved was one.

We conducted each search for the same queries for each of the three different trees. We varied the radius of each search, so that the searches returned 1 to 50 objects, as real-life use cases of metric indices usually only require a small part of the data set to be retrieved. For each radius, we conducted 100 searches, one for each test query. The average number of comparisons and the average number of objects retrieved of these 100 test queries for each index were saved as the result.

5.3 Implementation Details

As we only use the number of distance calculations performed as the performance measure in this report, programming language and hardware details are not necessary to reconstruct the experiments. Nevertheless, we will now present some details about our implementation as it is relevant for presenting the work behind this report.

5.3.1 Programming Language

The programming language used to conduct the experiments in this report was Julia, a free, open-source high-level language for scientific and technical computing [1]. We chose this language as it is reported to have benchmarking results relative to the C-language that are better than other high-level languages. Even though we did not measure CPU-time, we still preferred a fast programming language. This is because the data sets used in this report were relatively large, and all experiments were run on a MacBook with 1.1 GHz Intel Core M processor and 8 GB 1600 MHz DDR3 RAM. To be able to finish all experiments within the time frame of this project, a fast programming language was needed.

As we had no prior experience with programming in Julia, we had to spend some time getting to know the language. However, we found that the advantages of Julia outweighed the time spent learning the language in the initial phase of the project.

5.3.2 Technologies

Julia provides packages for some of the standard operations that were needed in this project. We used the package Distances [23] to calculate Euclidean distances between vectors, and the package Distributions [24] to generate Gaussian clusters. We used Hetland's implementation of Linear Program 3.1 to optimize the weights of the foci in the WSSS-tree, CSSS-tree, and W2SSS-tree, which uses the packages JuMP [22] and Clp [21]. We implemented the rest of the program, including the building and searching of all tree-structures, from scratch. To save and load built indices to and from memory, we used BSON [20]. To avoid having to recompile the entire program each time a small change was made, we used Revise [19].

The code that was needed to conduct experiments 1 and 2 in this report contained about 1000 lines of code, and the code that was needed to conduct Experiment 3 also contained about 1000 lines of code.

5.3.3 Method

We used pair programming when implementing the code in this project. For version control, we used GitHub, and our repository is found through this link: `https://github.com/helenemoe/masteroppgave/tree/delivery`.

Chapter 6

Experiment 1

This chapter will present Experiment 1 and its results. We tested how well the SSS-tree, WSSS-tree, and CSSS-tree perform on queries from one cluster.

6.1 Experiment Description

The purpose of this experiment was to test Hypothesis 1: *The WSSS-tree will outperform the SSS-tree if the queries come from one cluster.*

To test Hypothesis 1, we compared the result of the SSS-tree, the WSSS-tree, and the CSSS-tree. We included the CSSS-tree to act as a lower bound for the two other index structures. If the performance of the best of the SSS-tree and WSSS-tree is close to the performance of the CSSS-tree, it could indicate that this index structure is redundant.

6.1.1 Query selection

In this experiment, we used one cluster of queries for both training and test queries. This cluster was chosen by first selecting the data object that had the highest summed distance to all other objects in the data set, and then selecting the 150 objects that were closest to this object in the data set, including itself.

We chose a cluster center far away from the other objects in the data set, as this could theoretically give a higher margin to the hyperplane that separates the objects the region is responsible for and the training queries in the pivot space [17]. If the average distance from the queries to the other objects in the data set is large, this might cause fewer regions to overlap it, which could reduce the chance of the objective value of the linear program being zero.

6.2 Simulation Results

We conducted Experiment 1 on the eight different data sets described in Chapter 5 and the results are presented in the following subsections. For each data set, we present the difference in distance calculations in percent between the WSSS-tree and SSS-tree, and the WSSS-tree and the CSSS-tree. For the synthetic data sets, we also present the highest percentage of the data set that the WSSS-tree has to review, to see if we observe the curse of dimensionality.

6.2.1 Copenhagen Chromosome Data Set

The result of Experiment 1 on the Copenhagen Chromosome data set is found in Figure 6.1. Overall, the WSSS-tree conducts about 25–35% fewer distance calculations than the SSS-tree, when retrieving any number of objects smaller than 50. The performance of the WSSS-tree is almost the same as the performance of the CSSS-tree.

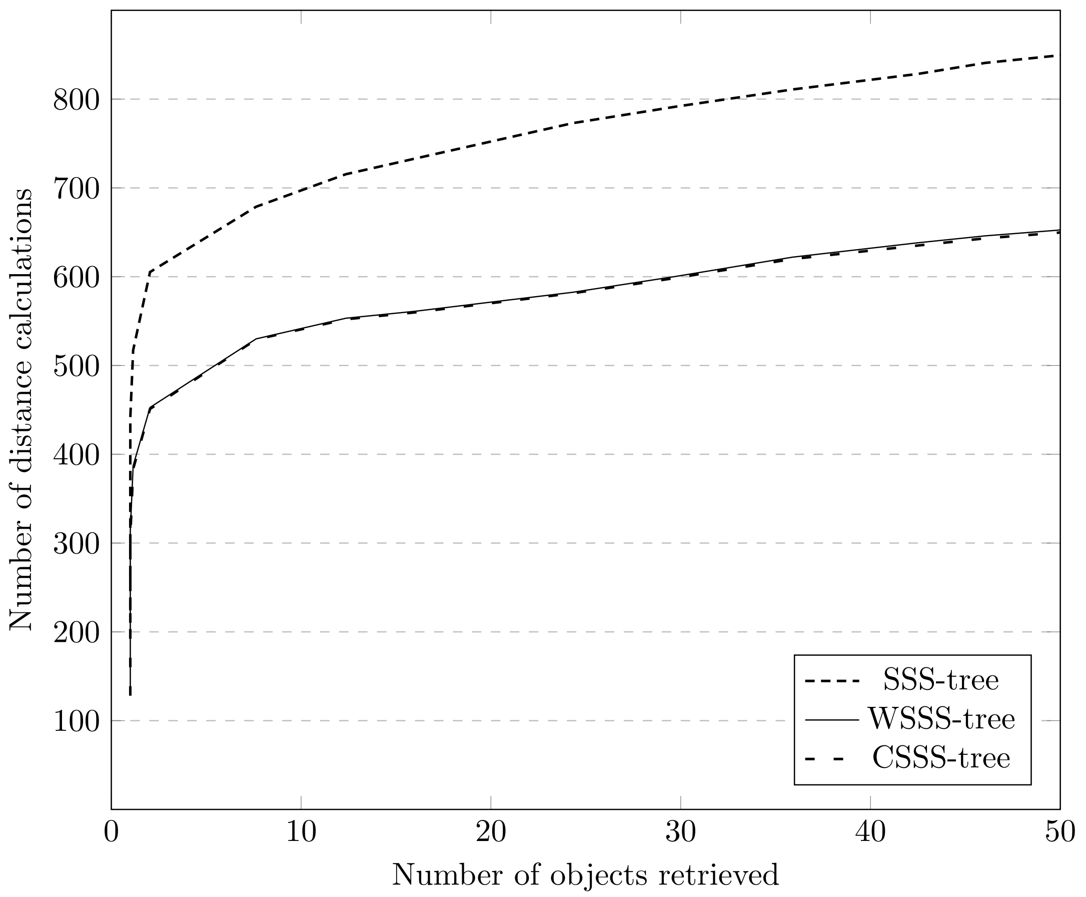


Figure 6.1: The number of distance calculations conducted by the WSSS-tree, the SSS-tree, and the CSSS-tree, when retrieving 1 to 50 objects by applying queries from one cluster to the Copenhagen Chromosome data set.

6.2.2 Corel Image Collection

The result of Experiment 1 on the Corel Image Collection is found in Figure 6.2. When retrieving only one object, the WSSS-tree conducts about 40% fewer distance calculations than the SSS-tree. As the number of objects returned gradually increases to 50, this percentage gradually decreases to about 25%. The performance of the WSSS-tree is almost on par with the performance of the CSSS-tree, with the WSSS-tree conducting about 5% more distance calculations.

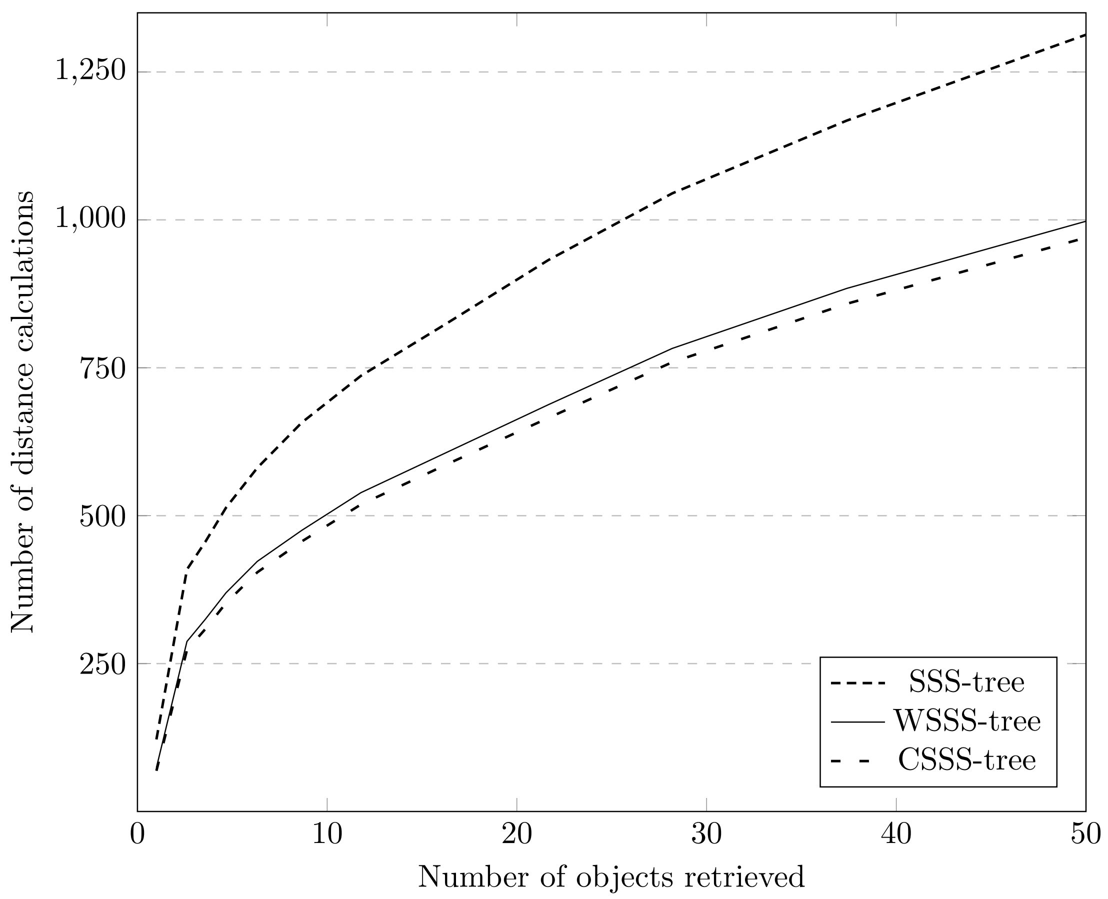


Figure 6.2: The number of distance calculations conducted by the WSSS-tree, the SSS-tree, and the CSSS-tree, when retrieving 1 to 50 objects by applying queries from one cluster to the Corel Image Collection.

6.2.3 Unitary Cube of Five Dimensions

The result of Experiment 1 on 100 000 synthetic random vectors in the unitary cube of five dimensions is found in Figure 6.3. When retrieving only one object, the WSSS-tree needs 50% fewer distance calculations than the SSS-tree. When retrieving closer to 50 objects, the WSSS-tree uses about 30% fewer distance calculations than the WSSS-tree. At most, the WSSS-tree reviews about 1% of the data set. The WSSS-tree performs only about 2% worse than the CSSS-tree overall.

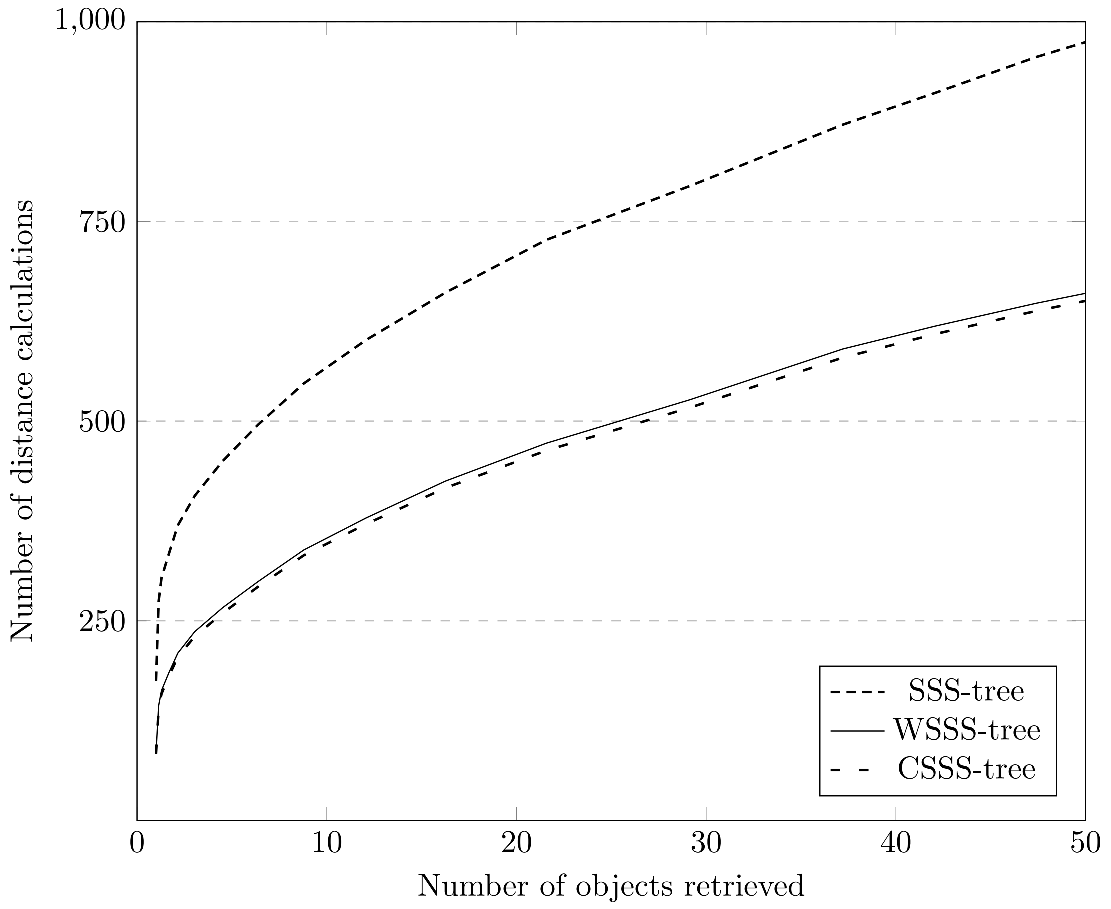


Figure 6.3: The number of distance calculations conducted by the WSSS-tree, the SSS-tree, and the CSSS-tree, when retrieving 1 to 50 objects by applying queries from one cluster to 100 000 synthetic random vectors in the unitary cube of five dimensions.

6.2.4 Gaussian Clusters of Five Dimensions

The result of Experiment 1 on a set of 100 000 vectors of five dimensions distributed as ten Gaussian clusters is found in Figure 6.4. The WSSS-tree is about 45% more efficient than the SSS-tree when retrieving only one object. This percentage decreases to around 30% when retrieving 50 objects. At most, the WSSS-tree reviews slightly more than 1% of the data set. The WSSS-tree performs only about 3% worse than the CSSS-tree overall.

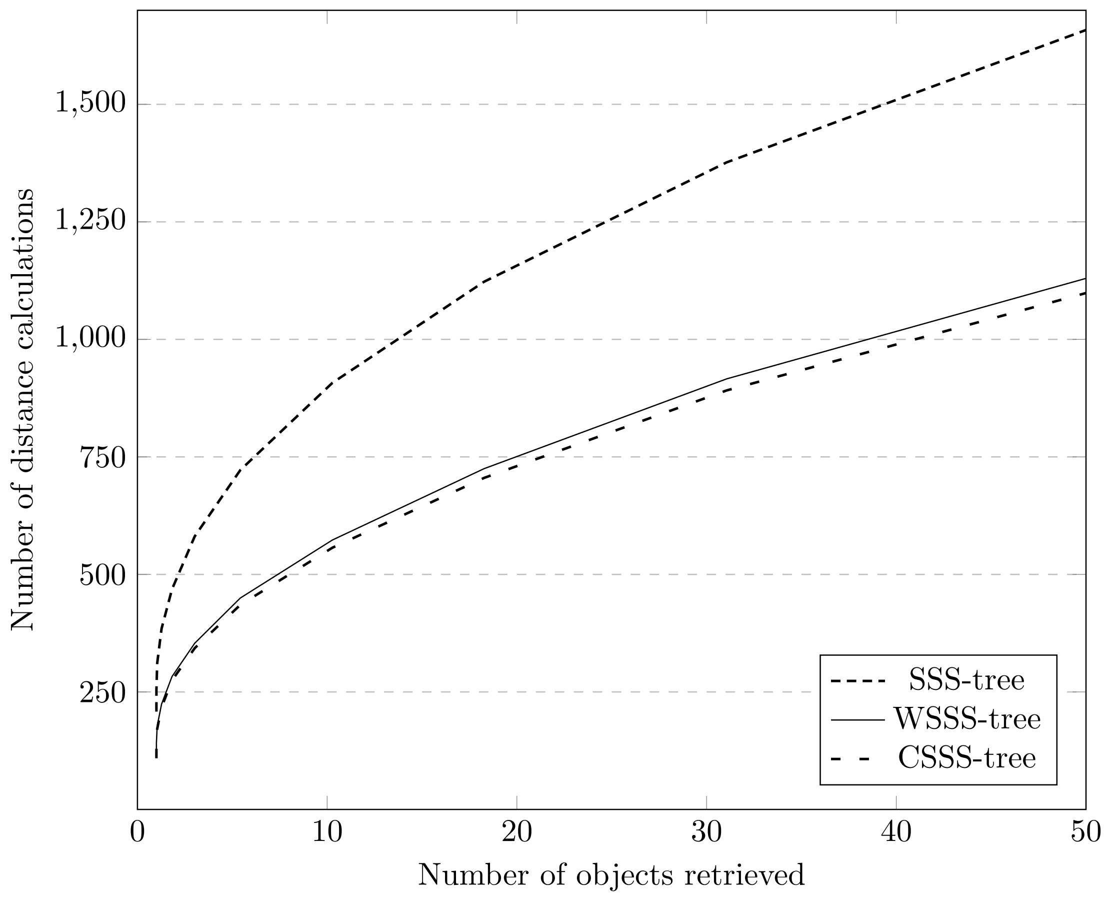


Figure 6.4: The number of distance calculations conducted by the WSSS-tree, the SSS-tree, and the CSSS-tree, when retrieving 1 to 50 objects by applying queries from one cluster to 100 000 vectors of five dimensions distributed as ten Gaussian clusters.

6.2.5 Unitary Cube of Ten Dimensions

The result of Experiment 1 on 100 000 synthetic random vectors in the unitary cube of ten dimensions is found in Figure 6.5. When retrieving only one object, the WSSS-tree needs 55% fewer distance calculations than the SSS-tree. When retrieving closer to 50 objects, the WSSS-tree reviews more than 25% fewer objects than the SSS-tree. At most, the WSSS-tree reviews about 11% of the data set. The WSSS-tree performs only about 1.5% worse than the CSSS-tree overall.

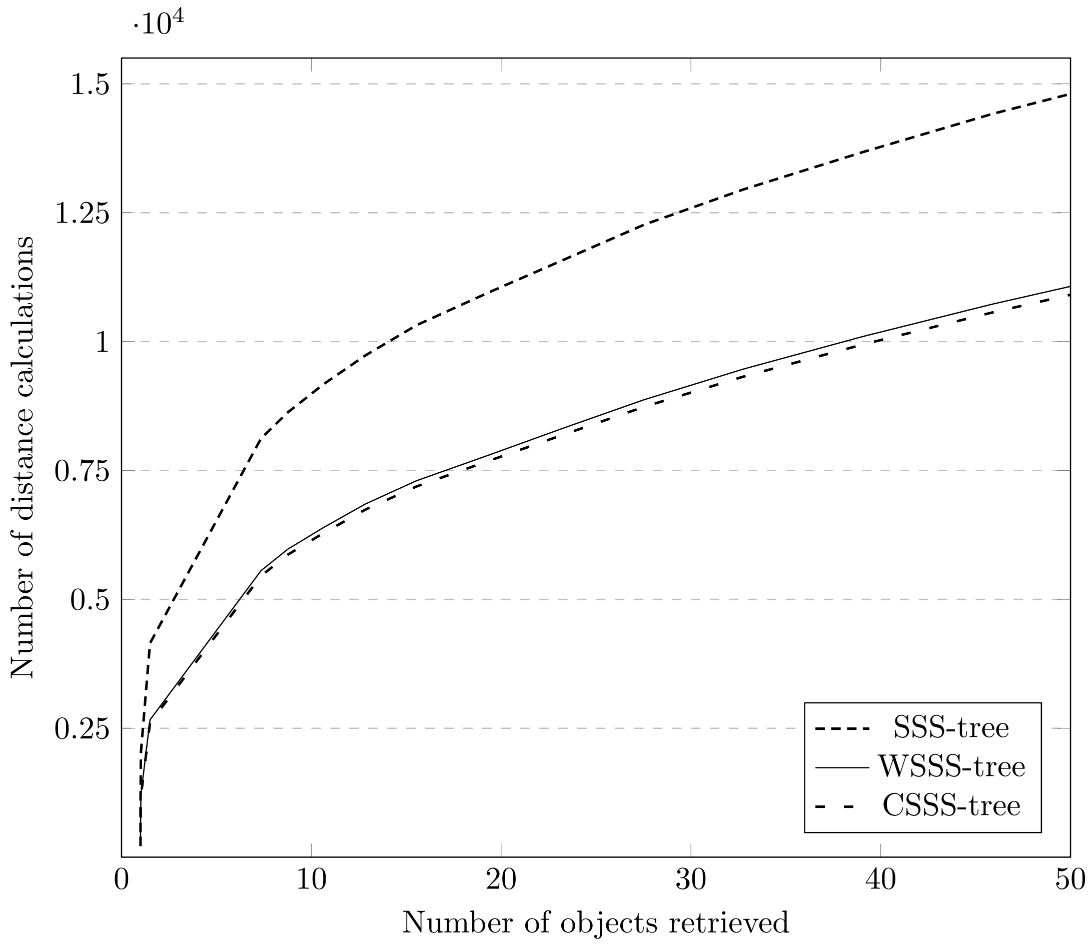


Figure 6.5: The number of distance calculations conducted by the WSSS-tree, the SSS-tree, and the CSSS-tree, when retrieving 1 to 50 objects by applying queries from one cluster to 100 000 synthetic random vectors in the unitary cube of ten dimensions.

6.2.6 Gaussian Clusters of Ten Dimensions

The result of Experiment 1 on a set of 100 000 vectors of ten dimensions distributed as ten Gaussian clusters is found in Figure 6.6. The WSSS-tree is 50% more efficient than the SSS-tree when retrieving only one object. When retrieving up to 50 objects, it performs almost 30% fewer distance calculations compared to the SSS-tree. The WSSS-tree reviews at most about 15% of the data set. Overall, the WSSS-tree performs only about 3% worse than the CSSS-tree.

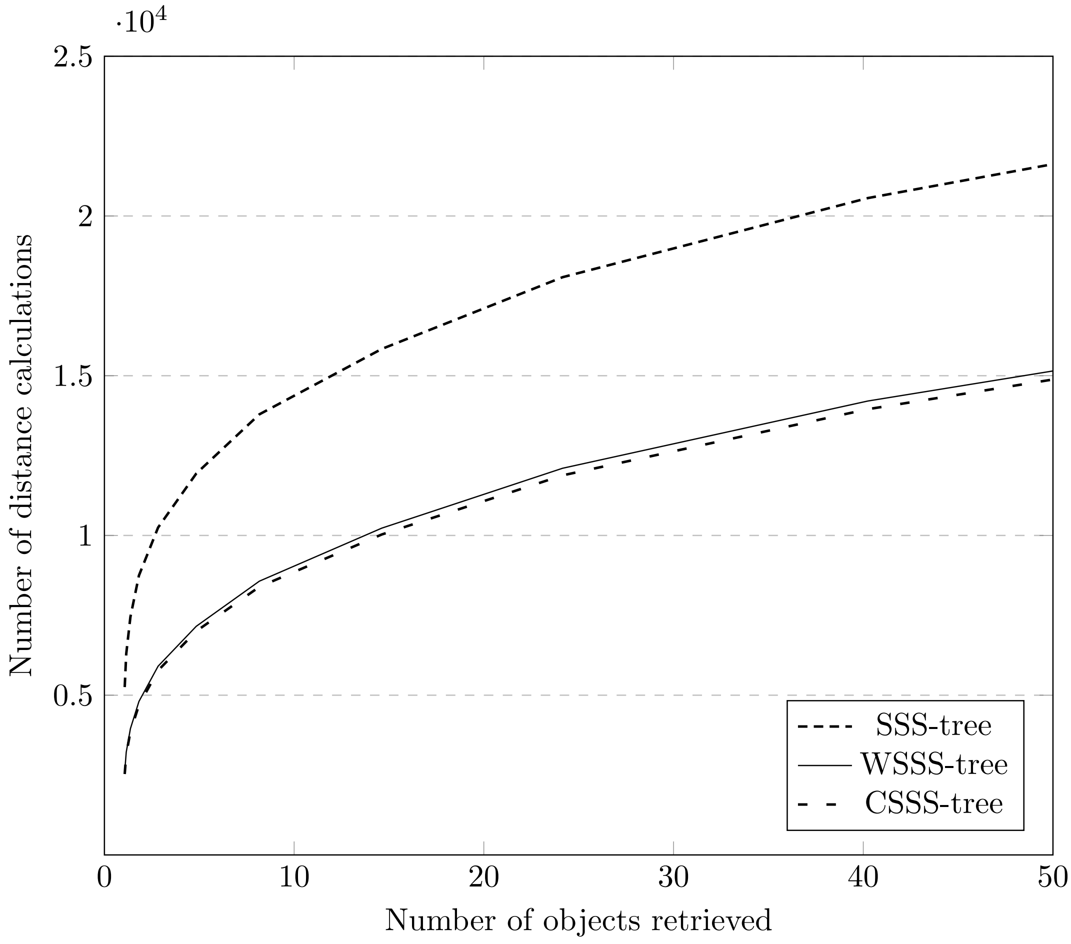


Figure 6.6: The number of distance calculations conducted by the WSSS-tree, the SSS-tree, and the CSSS-tree, when retrieving 1 to 50 objects by applying queries from one cluster to 100 000 vectors of ten dimensions distributed as ten Gaussian clusters.

6.2.7 Unitary Cube of 15 Dimensions

The result of Experiment 1 on 100 000 synthetic random vectors in the unitary cube of 15 dimensions is found in Figure 6.7. The WSSS-tree is about 45% better than the SSS-tree when retrieving one object, and when retrieving 50 objects, it is about 10% better. The WSSS-tree reviews almost 50% of the data set at most. The WSSS-tree performs only about 1.5% worse than the CSSS-tree overall.

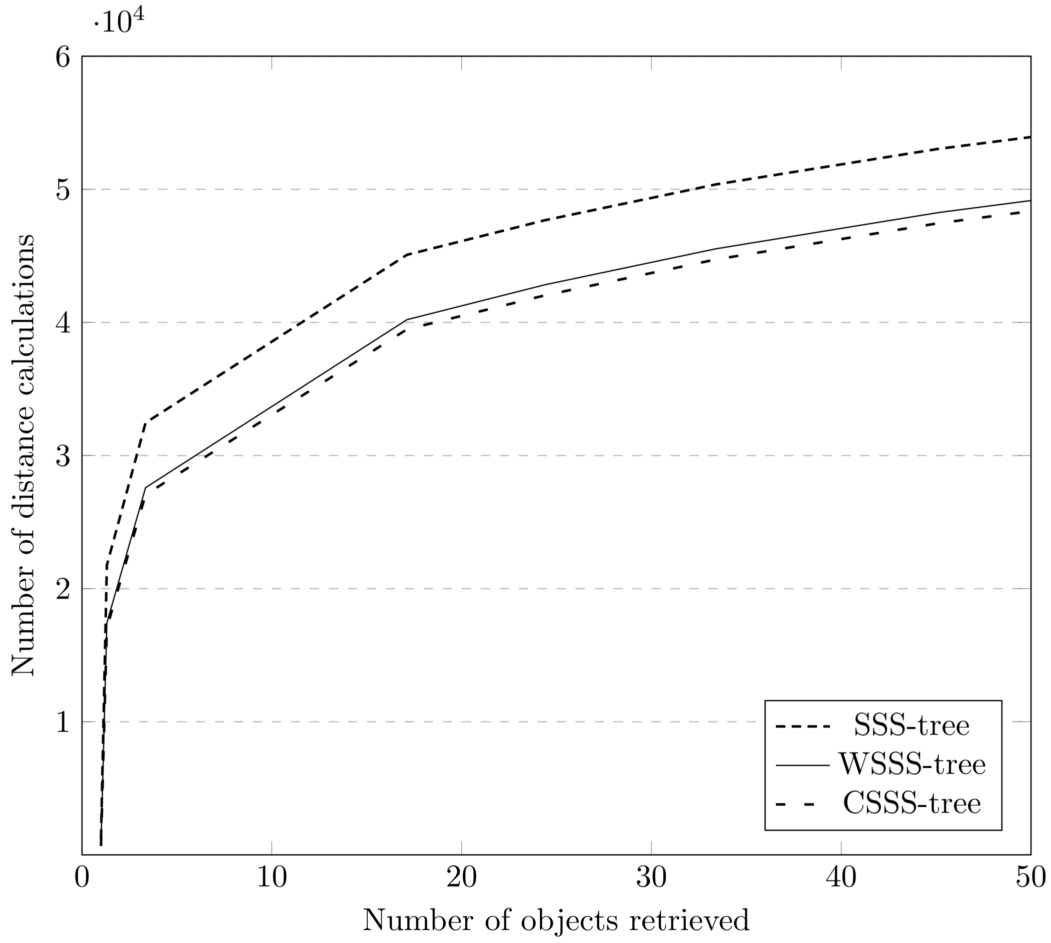


Figure 6.7: The number of distance calculations conducted by the WSSS-tree, the SSS-tree, and the CSSS-tree, when retrieving 1 to 50 objects by applying queries from one cluster to 100 000 synthetic random vectors in the unitary cube of 15 dimensions.

6.2.8 Gaussian Clusters of 15 Dimensions

The result of Experiment 1 on a set of 100 000 vectors of 15 dimensions distributed as ten Gaussian clusters is found in Figure 6.8. The WSSS-tree conducts about 25% fewer distance calculations than the SSS-tree when retrieving only one object. When retrieving more objects, the percentage decreases to about 10%. The WSSS-tree reviews around 55% of the data set at most. The WSSS-tree performs only about 2% worse than the CSSS-tree.

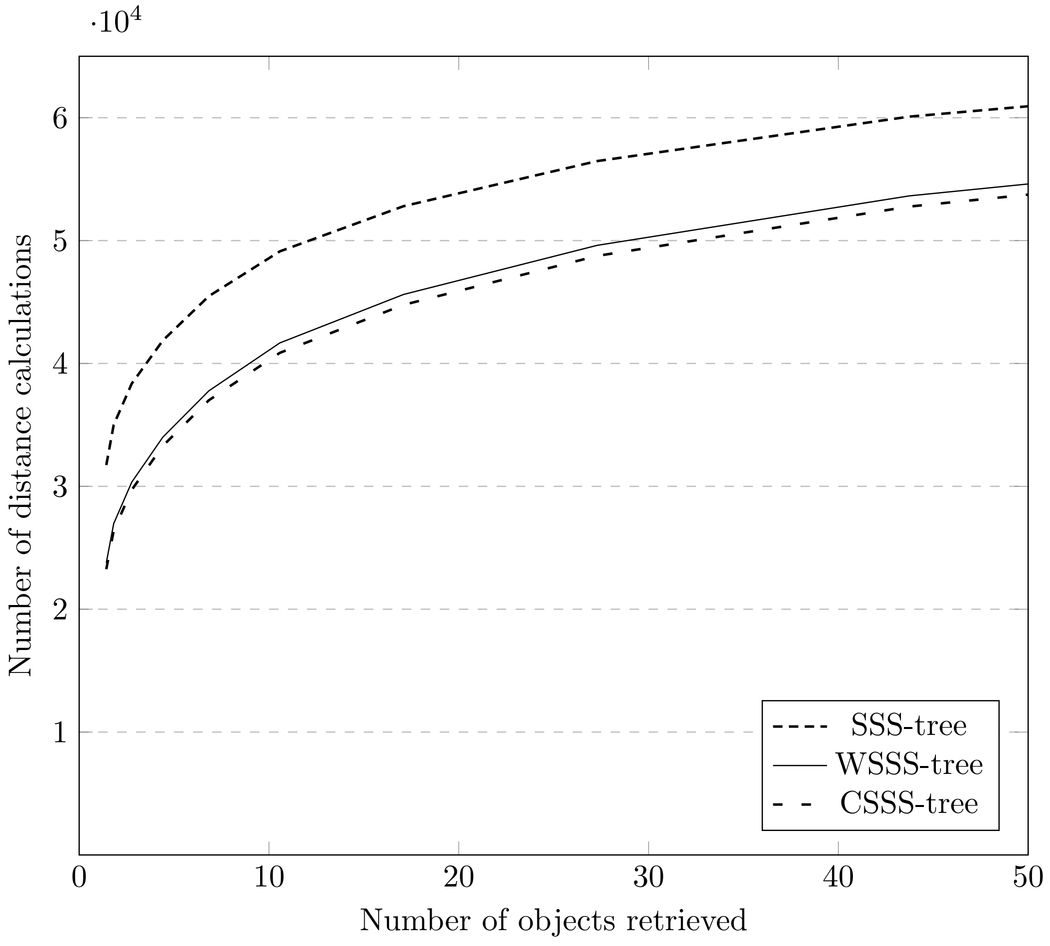


Figure 6.8: The number of distance calculations conducted by the WSSS-tree, the SSS-tree, and the CSSS-tree, when retrieving 1 to 50 objects by applying queries from one cluster to 100 000 vectors of 15 dimensions distributed as ten Gaussian clusters.

6.3 Summary and Discussion of the Results

For all the metric spaces used in this experiment, the WSSS-tree needs significantly fewer distance calculations than the SSS-tree. The most significant difference is found when retrieving one object from random vectors in the unitary cube of ten dimensions, where the WSSS-tree conducts 55% fewer distance calculations than the SSS-tree. The SSS-tree has outperformed other clustering-based metric indexing methods in this metric space, which suggests that the WSSS-tree also could outperform these indexing methods in this metric space. Common for all the data sets is that the difference in percentage decreases when the number of objects retrieved increases. This percentage is still never below 25%, except for the generated data sets of 15 dimensions, where this percentage is 10. However, for these two data sets, more than half of the data set is reviewed when retrieving 50 objects. When such a large part of the data set is reviewed, one risks that the amount of overhead involved with the index structure will surpass the reduction in distance calculations and, through this, cause the performance of the index to be worse than the performance of a linear scan. This suggests that the WSSS-tree does not overcome the curse of dimensionality.

The CSSS-tree always performs better than both the SSS-tree and the WSSS-tree. This is expected as it is designed to work as a lower bound for both index structures. However, the difference between the WSSS-tree and the CSSS-tree is marginal, and at most 5%. This could indicate that the CSSS-tree is redundant for the scenario used in this experiment.

The observations in these metric spaces support Hypothesis 1.

Chapter 7

Experiment 2

This chapter will present Experiment 2 and its results. In this experiment, we tested how well the SSS-tree, WSSS-tree, and CSSS-tree perform on queries from two clusters.

7.1 Experiment Description

The purpose of this experiment was to test Hypothesis 2: *The WSSS-tree will lose a part of its advantage over the SSS-tree when the queries come from more than one cluster if the clusters are so far apart that they would not be classified as one larger cluster.*

To do this, we applied queries from two clusters to the SSS-tree, the WSSS-tree, and the CSSS-tree, and compared the results. We included the CSSS-tree to act as a lower bound for the SSS-tree and WSSS-tree.

We conducted preliminary experiments with queries from up to five clusters. However, we only included the results of the experiments with queries from two clusters, as we saw little difference in the results. Still, further research could be done on this.

7.1.1 Query selection

In this experiment, we used both training and test queries from two clusters. To be able to test Hypothesis 2, these clusters must be significantly different. To ensure this, we chose the clusters the following way:

- The first cluster was chosen the same way as in Experiment 1. This cluster was chosen by first selecting the object that has the biggest summed distance to all other objects in the data set and then selecting the 75 objects in the data set closest to this object, including itself.

- The second cluster was chosen by choosing the object that has the biggest summed distance to the objects in the already selected cluster, then selecting the 75 objects in the data set closest to this object, including itself.

7.2 Simulation Results

We conducted Experiment 2 on the same eight data sets as we conducted Experiment 1 on, described in Chapter 5. The results are presented in the following subsections. For each data set, we present the same statistics as in Experiment 1.

7.2.1 Copenhagen Chromosome Data Set

The result of Experiment 2 on the Copenhagen Chromosome data set is found in Figure 7.1. The WSSS-tree overall conducts about 0.5–3% fewer distance calculations than the SSS-tree for each number of objects retrieved. The performance of the CSSS-tree is better than the performance of the WSSS-tree and the SSS-tree, but neither ever conducts more than 5% more distance calculations than the CSSS-tree.

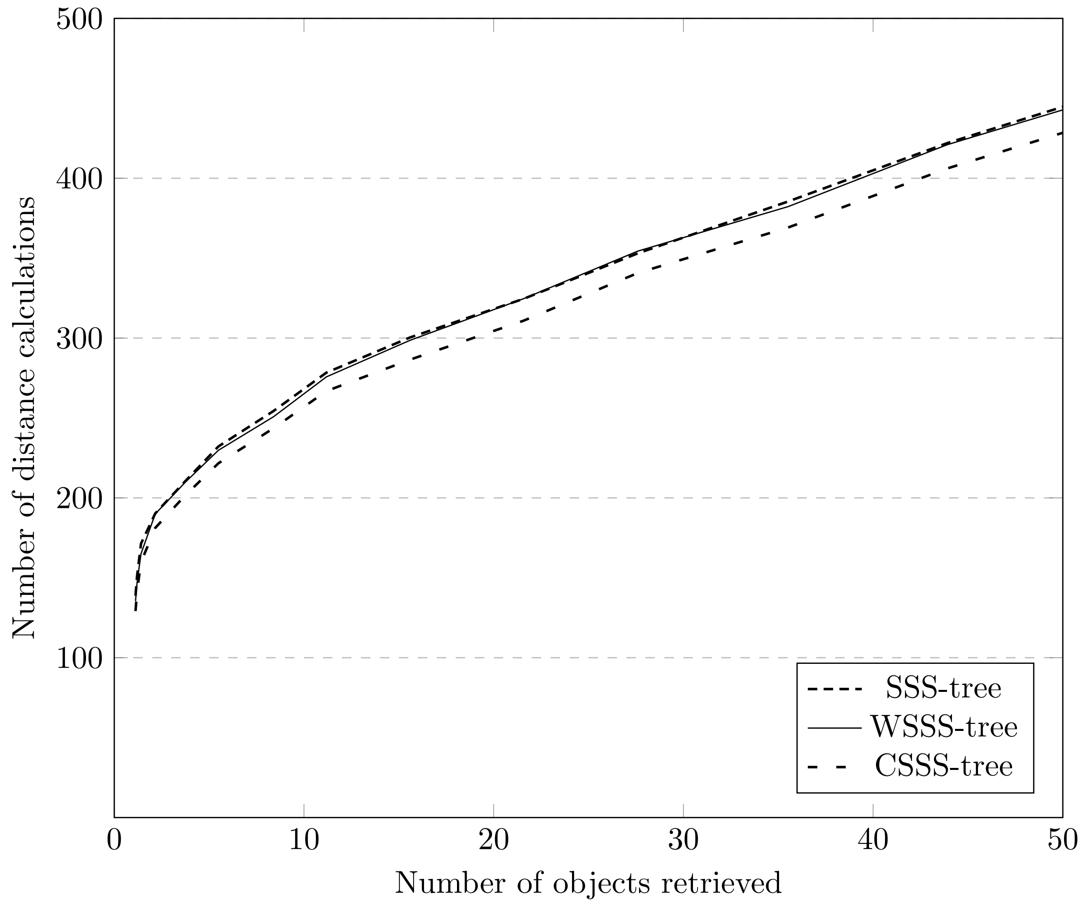


Figure 7.1: The number of distance calculations conducted by the WSSS-tree, the SSS-tree, and the CSSS-tree, when retrieving 1 to 50 objects by applying queries from two clusters to the Copenhagen Chromosome data set.

7.2.2 Corel Image Collection

The result of Experiment 2 on the Corel Image Collection is found in Figure 7.2. When retrieving 1 to 50 objects, the WSSS-tree conducts about 5–10% fewer distance calculations than the SSS-tree. The WSSS-tree conducts about 5–7% more distance calculations than the CSSS-tree.

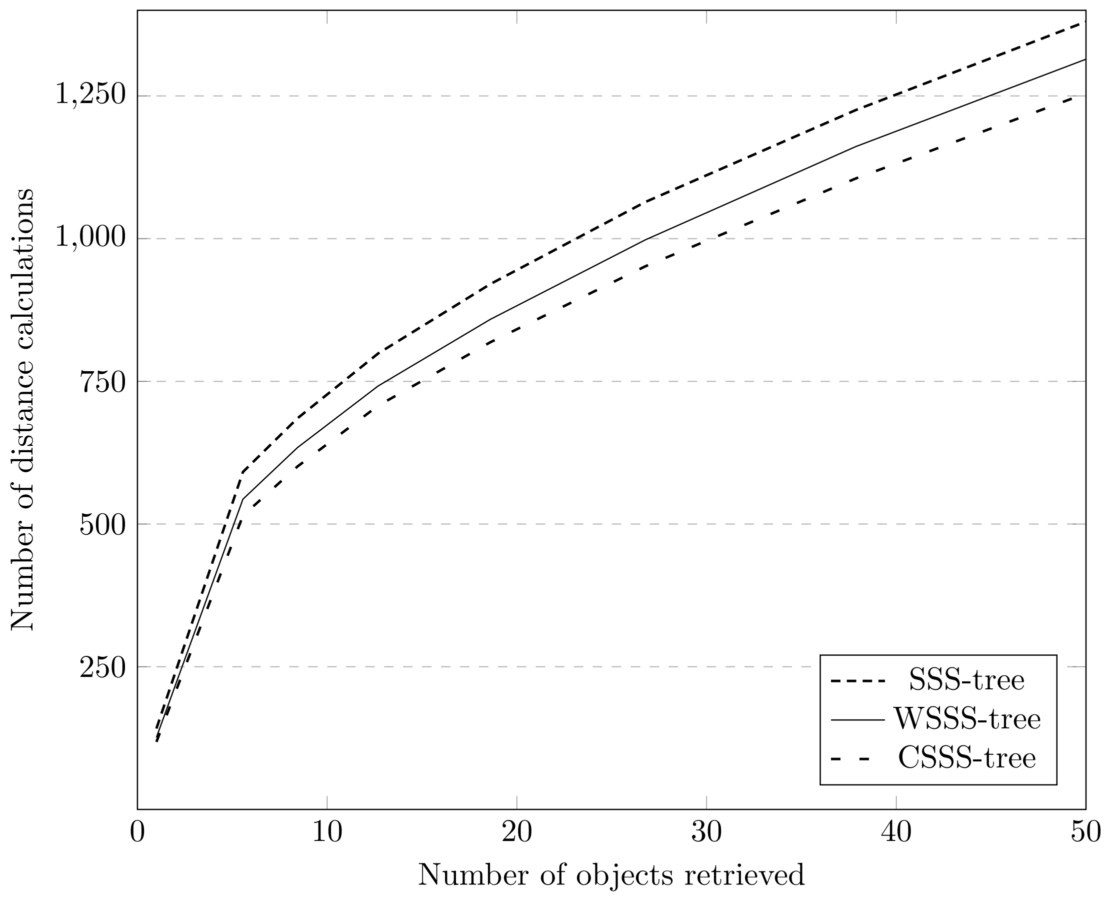


Figure 7.2: The number of distance calculations conducted by the WSSS-tree, the SSS-tree, and the CSSS-tree, when retrieving 1 to 50 objects by applying queries from two clusters to the Corel Image Collection.

7.2.3 Unitary Cube of Five Dimensions

The result of Experiment 2 on 100 000 synthetic random vectors in the unitary cube of five dimensions is found in Figure 7.3. When retrieving only one object, the WSSS-tree needs 15% fewer distance calculations than the SSS-tree. When retrieving closer to 50 objects, the WSSS-tree is almost 10% better than the SSS-tree. At most, the WSSS-tree reviews less than 1% of the data set. The WSSS-tree is only about 3–4% worse than the CSSS-tree overall.

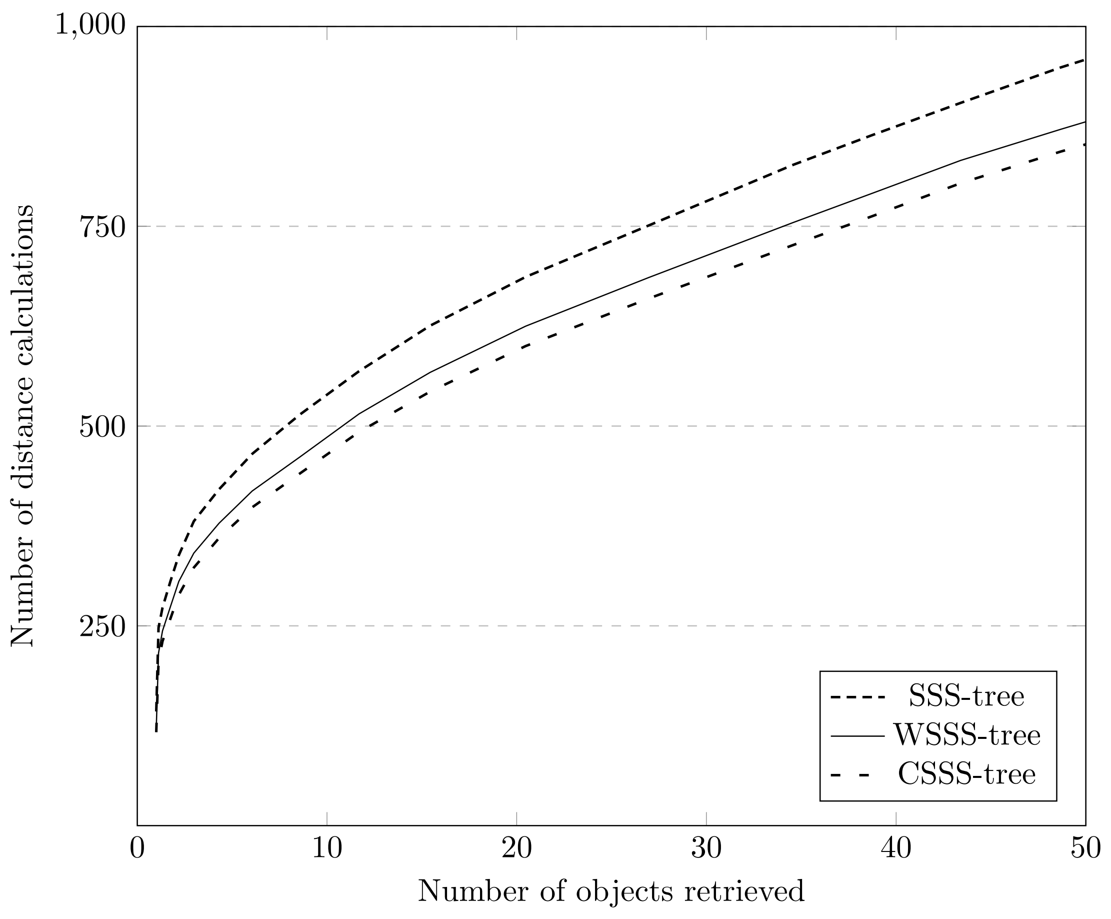


Figure 7.3: The number of distance calculations conducted by the WSSS-tree, the SSS-tree, and the CSSS-tree, when retrieving 1 to 50 objects by applying queries from two clusters to 100 000 synthetic random vectors in the unitary cube of five dimensions.

7.2.4 Gaussian Clusters of Five Dimensions

The result of Experiment 2 on a set of 100 000 vectors of five dimensions distributed as ten Gaussian clusters is found in Figure 7.4. The WSSS-tree is about 20% better than the SSS-tree when retrieving only one object. This percentage decreases to almost 10% when retrieving 50 objects. At most, the WSSS-tree reviews about 1.4% of the data set. The WSSS-tree is about 4–5% worse than the CSSS-tree overall.

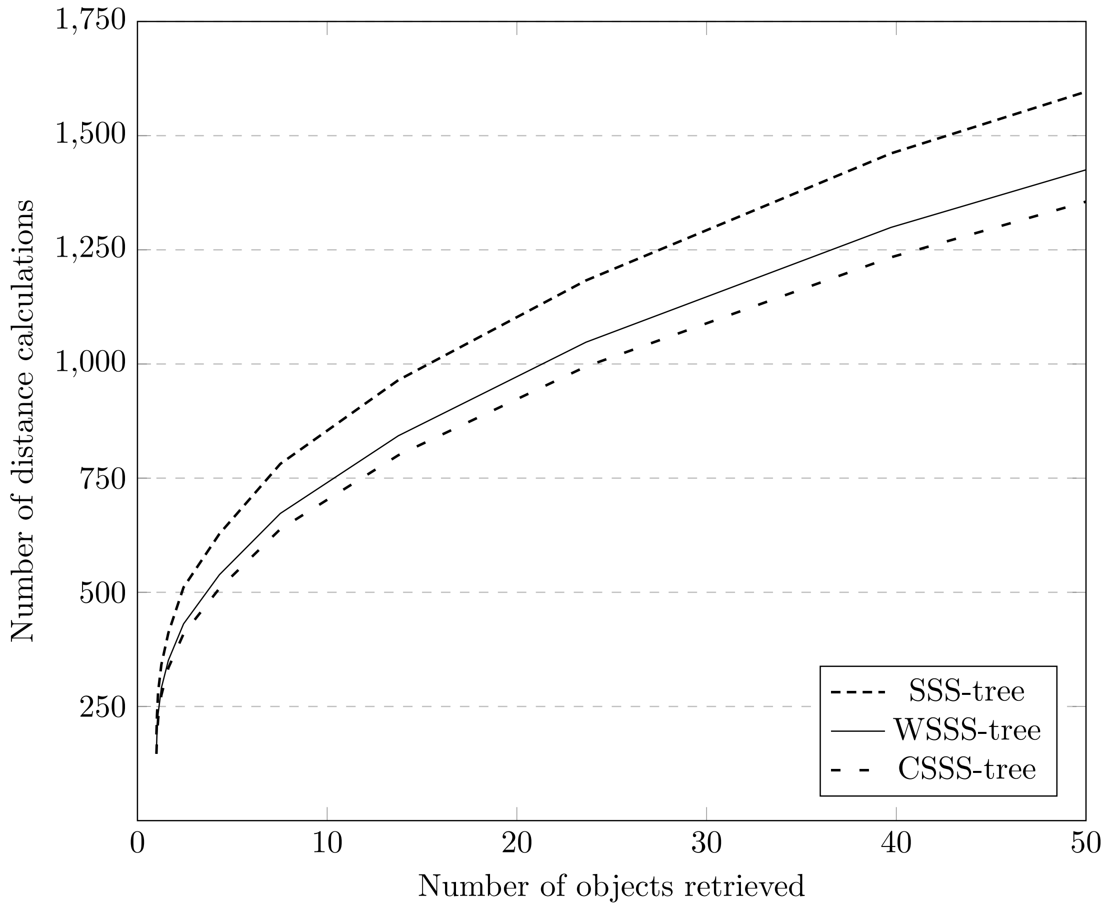


Figure 7.4: The number of distance calculations conducted by the WSSS-tree, the SSS-tree, and the CSSS-tree, when retrieving 1 to 50 objects by applying queries from two clusters to 100 000 vectors of five dimensions distributed as ten Gaussian clusters.

7.2.5 Unitary Cube of Ten Dimensions

The result of Experiment 2 on 100 000 synthetic random vectors in the unitary cube of ten dimensions is found in Figure 7.5. When retrieving only one object, the WSSS-tree conducts about 20% fewer distance calculations than the SSS-tree. When retrieving closer to 50 objects, the WSSS-tree is about 10% better than the SSS-tree. At most, the WSSS-tree reviews about 13% of the data set. The WSSS-tree is about 0.5–3% worse than the CSSS-tree overall.

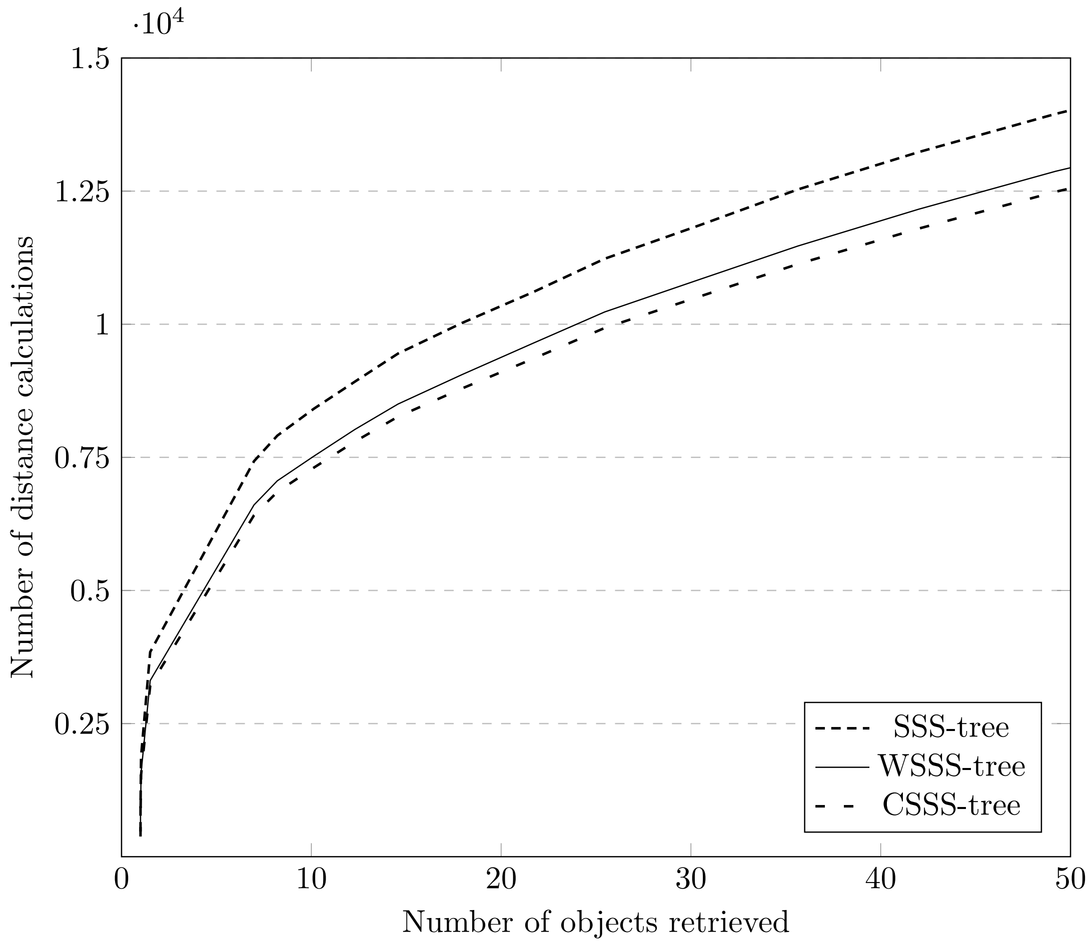


Figure 7.5: The number of distance calculations conducted by the WSSS-tree, the SSS-tree, and the CSSS-tree, when retrieving 1 to 50 objects by applying queries from two clusters to 100 000 synthetic random vectors in the unitary cube of ten dimensions.

7.2.6 Gaussian Clusters of Ten Dimensions

The result of Experiment 2 on a set of 100 000 vectors of ten dimensions distributed as ten Gaussian clusters is found in Figure 7.6. The performance of the WSSS-tree is about 20% better than the SSS-tree when retrieving only one object. When retrieving up to 50 objects, it performs about 10% fewer distance calculations compared to the SSS-tree. The WSSS-tree reviews at most less than 20% of the data set. Overall, the WSSS-tree is about 5% worse than the

CSSS-tree.

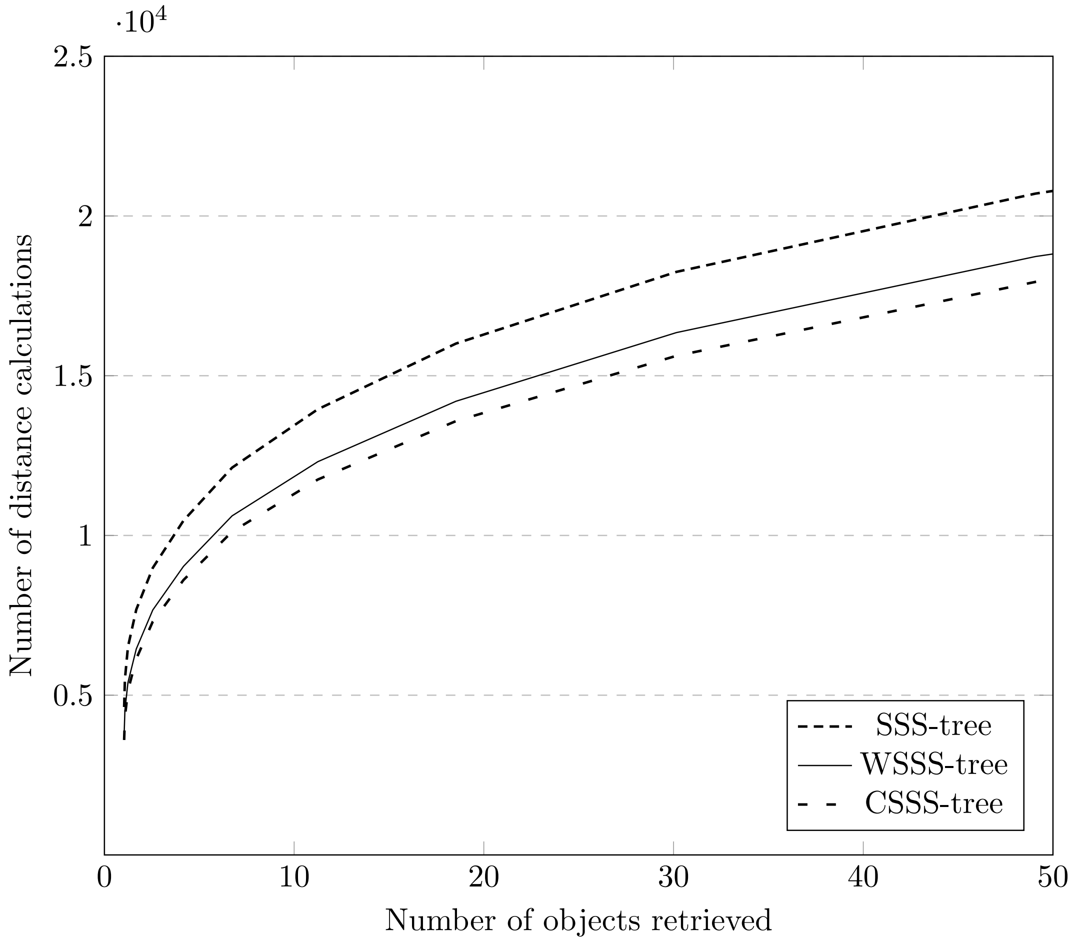


Figure 7.6: The number of distance calculations conducted by the WSSS-tree, the SSS-tree, and the CSSS-tree, when retrieving 1 to 50 objects by applying queries from two clusters to 100 000 vectors of ten dimensions distributed as ten Gaussian clusters.

7.2.7 Unitary Cube of 15 Dimensions

The result of Experiment 2 on 100 000 synthetic random vectors in the unitary cube of 15 dimensions is found in Figure 7.7. The performance of the WSSS-tree is about 20% better than the SSS-tree performance when retrieving one object, and when retrieving 50 objects, it is about 3% better. The WSSS-tree reviews more than 50% of the data set at most. The WSSS-tree is about 1–2% worse than the CSSS-tree overall.

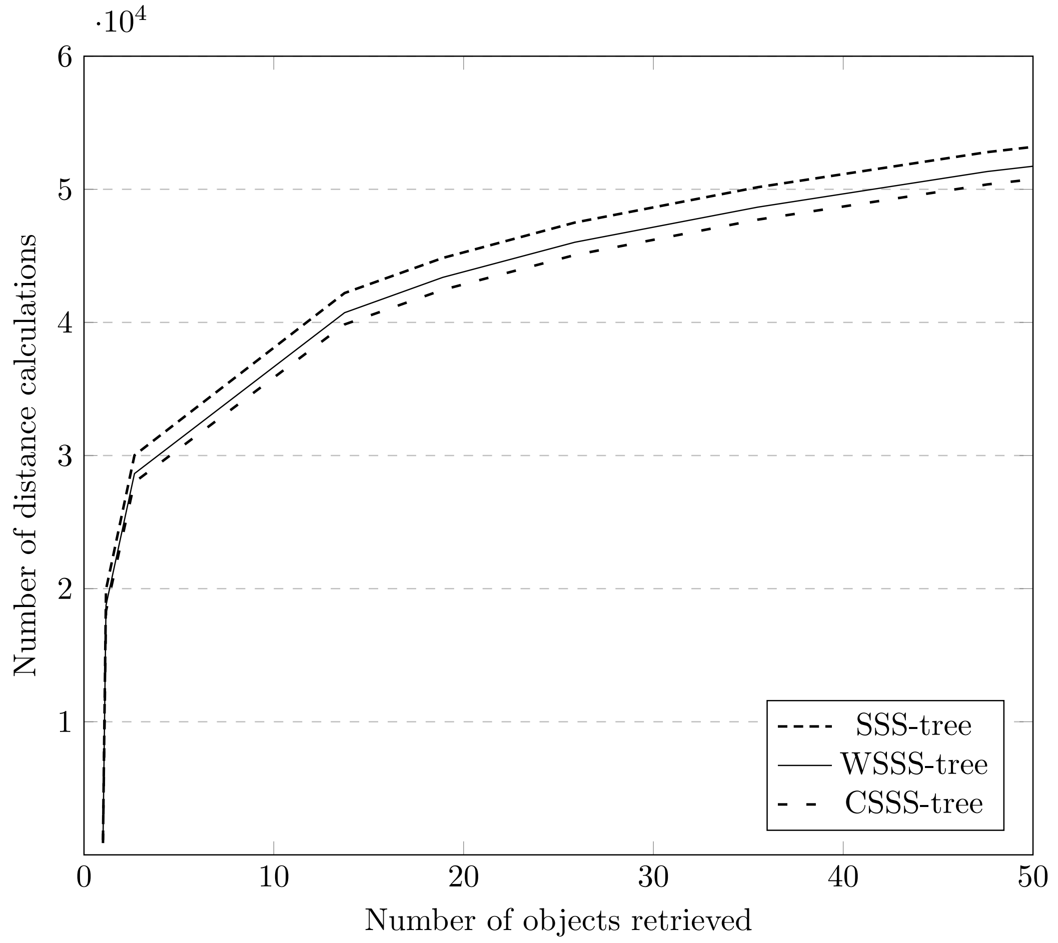


Figure 7.7: The number of distance calculations conducted by the WSSS-tree, the SSS-tree, and the CSSS-tree, when retrieving 1 to 50 objects by applying queries from two clusters to 100 000 synthetic random vectors in the unitary cube of 15 dimensions.

7.2.8 Gaussian Clusters of 15 Dimensions

The result of Experiment 2 on a set of 100 000 vectors of 15 dimensions distributed as ten Gaussian clusters is found in Figure 7.8. The WSSS-tree conducts about 10% fewer distance calculations than the SSS-tree when retrieving only one object. When retrieving 50 objects, the percentage decreases to about 3%. The WSSS-tree reviews at most more than 55% of the data set. The WSSS-tree is about 2–3% worse than the CSSS-tree.

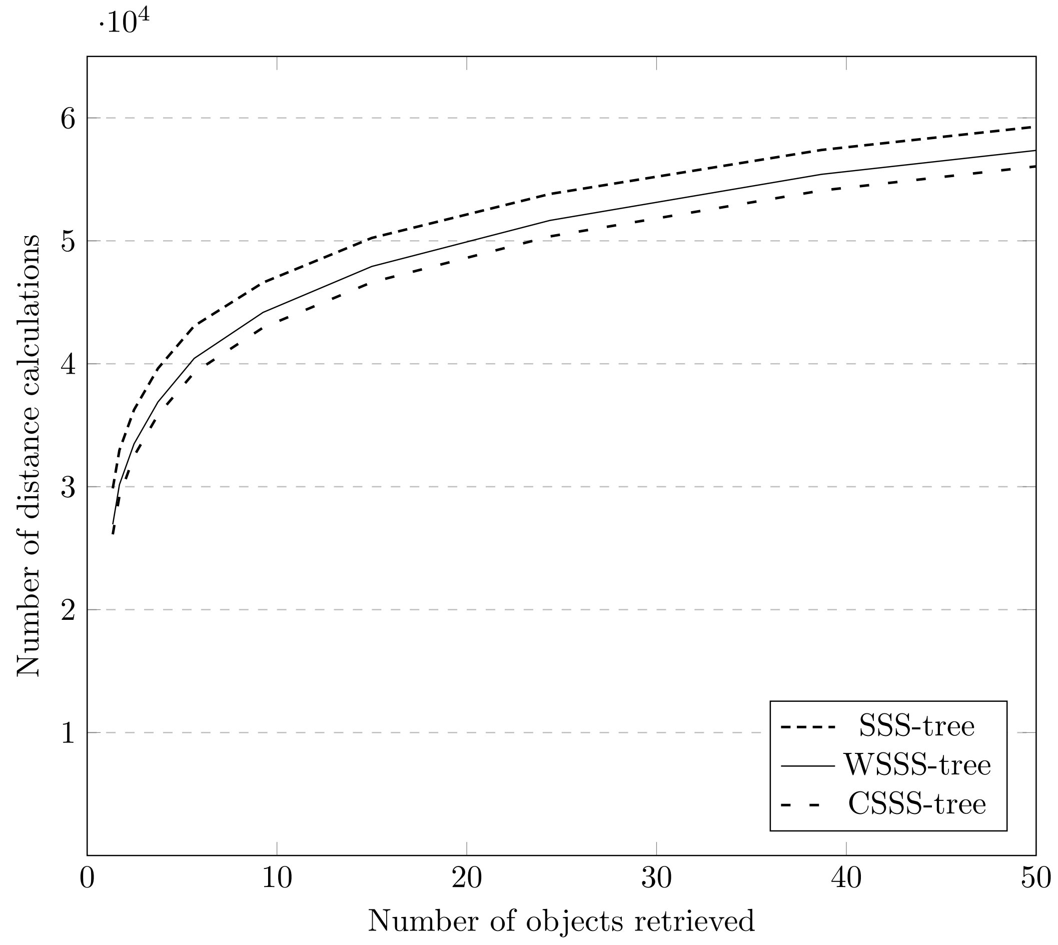


Figure 7.8: The number of distance calculations conducted by the WSSS-tree, the SSS-tree, and the CSSS-tree, when retrieving 1 to 50 objects by applying queries from two clusters to 100 000 vectors of 15 dimensions distributed as ten Gaussian clusters.

7.3 Summary and Discussion of the Results

In this experiment, the difference between the WSSS-tree and the SSS-tree is significantly less than in Experiment 1 for all metric spaces used. At most, the WSSS-tree conducts about 20% fewer distance calculations than the SSS-tree, compared to 55% in Experiment 1. In this experiment, it is the synthetic data sets of 10 dimensions, and the random vectors from the unitary cube of 15 dimensions that have the most significant advantage. In Experiment 1, the random vectors from the unitary cube of ten dimensions also have the most significant difference in performance, which is the metric space that the SSS-tree is shown to outperform other clustering-based indices in.

For the Copenhagen Chromosome data set, the difference in performance between the WSSS-tree and SSS-tree is at its lowest at 0.5%. In Experiment 1, the lowest percentage is never lower than 10%. The CSSS-tree has an advantage over the WSSS-tree varying from 0.5% to 7%, meaning that this advantage is comparable to the advantage the WSSS-tree has over the SSS-tree. This suggests that the CSSS-tree is not redundant for the scenario tested in this experiment.

From these observations, it seems as the WSSS-tree has lost parts of its advantage over the SSS-tree compared to in Experiment 1, supporting Hypothesis 2.

Chapter 8

Experiment 3

This chapter will present Experiment 3 and its results. In this experiment, we tested how well the W2SSS-tree perform on queries from two clusters, and compared it to the performance of the SSS-tree and the WSSS-tree.

8.1 Experiment Description

The purpose of this experiment was to test Hypothesis 3: *The W2SSS-tree will outperform both the SSS-tree and the WSSS-tree when the queries come from two clusters if the clusters are so far apart that they would not be classified as one larger cluster.*

To test Hypothesis 3, we compared the result of the SSS-tree, the WSSS-tree, and the W2SSS-tree. We wanted to see if the W2SSS-tree could gain an advantage over the WSSS-tree comparable to the advantage the WSSS-tree had over the SSS-tree in Experiment 1, which was between 10% and 55%. Therefore, we did not include a combined version of all three trees as the CSSS-tree did not cause an improvement in the performance of more than 7% in the previous experiments.

8.1.1 Query Selection

As in Experiment 2, the indices were built using 50 training queries and tested on 100 test queries. The queries came from two clusters and were selected the same way as in Experiment 2. The queries were chosen so that 25 training queries and 50 test queries were from the same cluster. For this experiment, the 25 training queries from the first cluster were used as *trainingQueries1*, and the 25 training queries from the second cluster were used as *trainingQueries2* when building the W2SSS-tree as described in Algorithm 7.

8.2 Simulation Results

We conducted Experiment 3 on the same eight data sets as we conducted experiments 1 and 2 on, described in Chapter 5. The results are presented in the following subsections. For each data set, we will present the difference in distance calculations in percent between the W2SSS-tree and SSS-tree, and between the W2SSS-tree and the WSSS-tree. For the synthetic data sets, we will also present the maximum percentage of the data set that is reviewed by the W2SSS-tree, to see if it can overcome the curse of dimensionality.

8.2.1 Copenhagen Chromosome Data Set

The result of Experiment 3 on the Copenhagen Chromosome data set is found in Figure 8.1. The W2SSS-tree conducts about 35% fewer distance calculations than the SSS-tree when retrieving only one object. When retrieving 50 objects, this percentage decreases to around 15%. The performance of the WSSS-tree is about the same as the SSS-tree. This means that the difference between the W2SSS-tree and the WSSS-tree is almost the same as the difference between the W2SSS-tree and the SSS-tree.

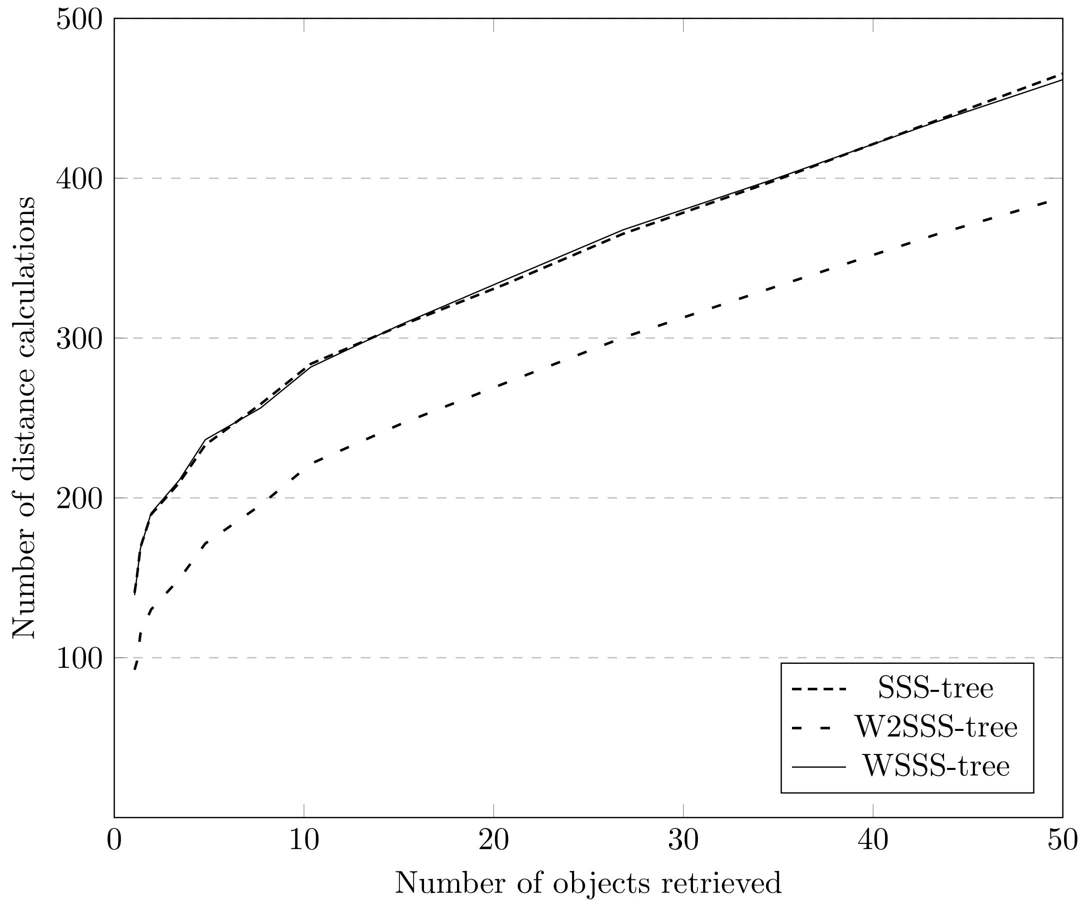


Figure 8.1: The number of distance calculations conducted by the WSSS-tree, the SSS-tree, and the W2SSS-tree, when retrieving 1 to 50 objects by applying queries from two clusters to the Copenhagen Chromosome data set.

8.2.2 Corel Image Collection

The result of Experiment 3 on the Corel Image Collection is found in Figure 8.2. When retrieving only one object, the W2SSS-tree conducts almost 50% fewer distance calculations than the SSS-tree and almost 45% fewer distance calculations than the WSSS-tree. As the number of objects returned gradually increases to 50, this percentage decreases to about 30% when compared to the SSS-tree and 25% when compared to the WSSS-tree.

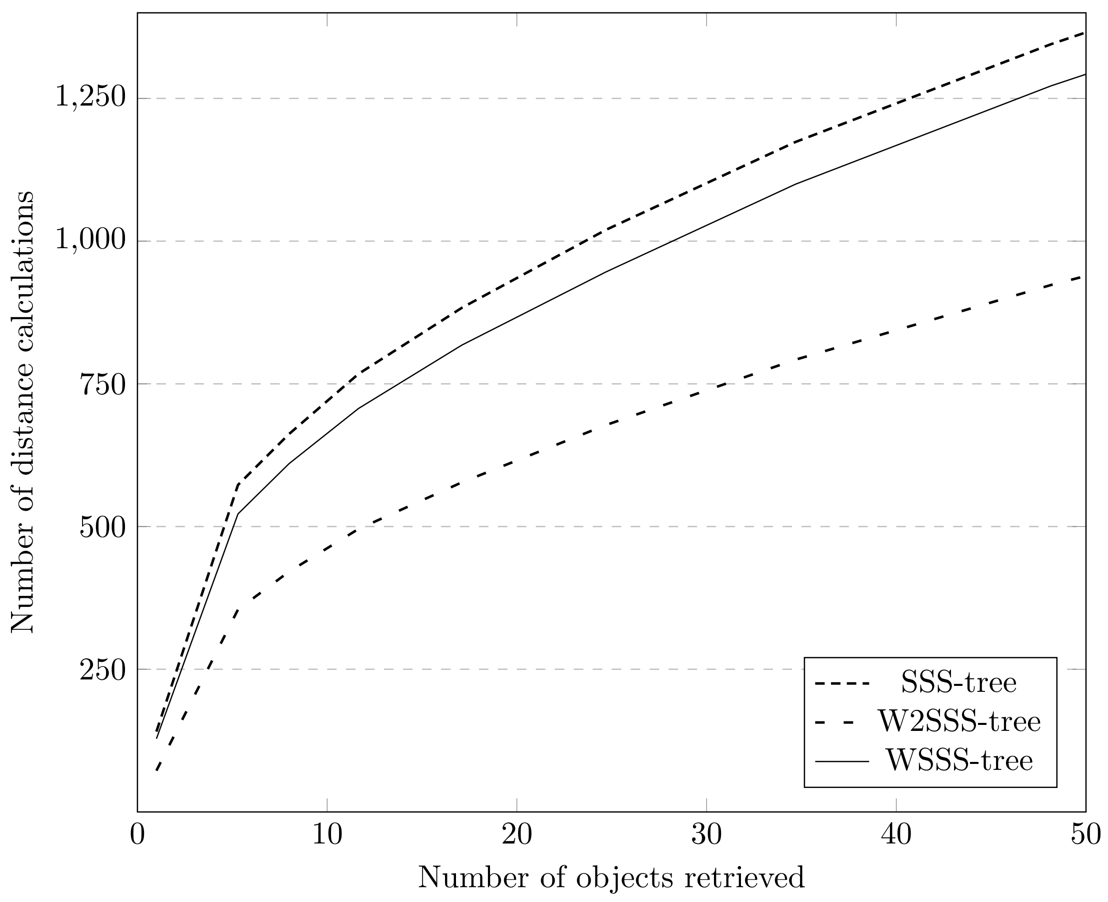


Figure 8.2: The number of distance calculations conducted by the W2SSS-tree, the SSS-tree, and the WSSS-tree, when retrieving 1 to 50 objects by applying queries from two clusters to the Corel Image Collection.

8.2.3 Unitary Cube of Five Dimensions

The result of Experiment 3 on 100 000 synthetic random vectors in the unitary cube of five dimensions is found in Figure 8.3. The W2SSS-tree needs 50% fewer distance calculations than the SSS-tree and 40% fewer distance calculations than the WSSS-tree when retrieving only one object. When retrieving closer to 50 objects, the W2SSS-tree uses about 30% fewer distance calculations than the SSS-tree, and 25% fewer distance calculations than the WSSS-tree. The W2SSS-tree reviews at most 0.6% of the data set.

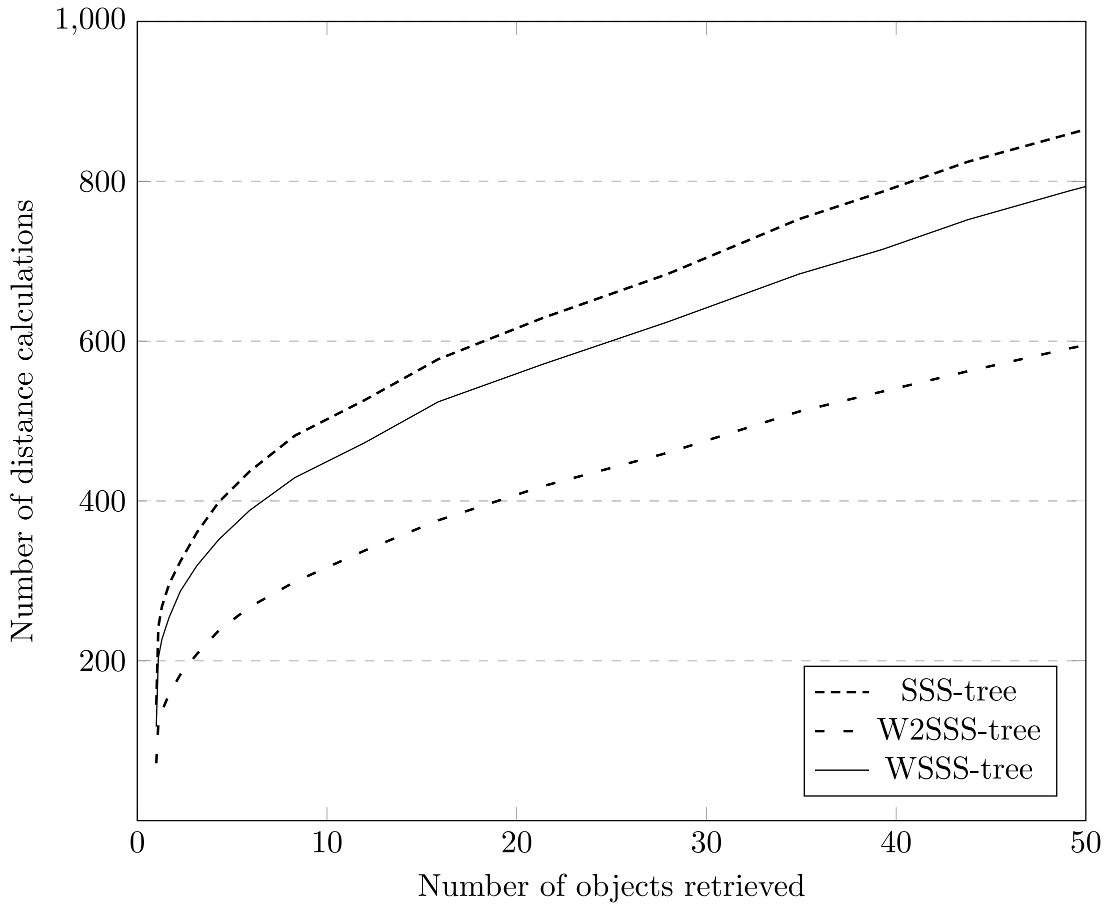


Figure 8.3: The number of distance calculations conducted by the SSS-tree, the W2SSS-tree, and the WSSS-tree, when retrieving 1 to 50 objects by applying queries from two clusters to 100 000 synthetic random vectors in the unitary cube of five dimensions.

8.2.4 Gaussian Clusters of Five Dimensions

The result of Experiment 3 on a set of 100 000 vectors of five dimensions distributed as ten Gaussian clusters is found in Figure 8.4. The W2SSS-tree is about 55% more efficient than the SSS-tree, and 45% more efficient than the WSSS-tree, when retrieving only one object. This percentage decreases to around 40% when compared to the SSS-tree, and 30% when compared to the WSSS-tree, when retrieving 50 objects. At most, the W2SSS-tree reviews slightly more than 1% of the data set.

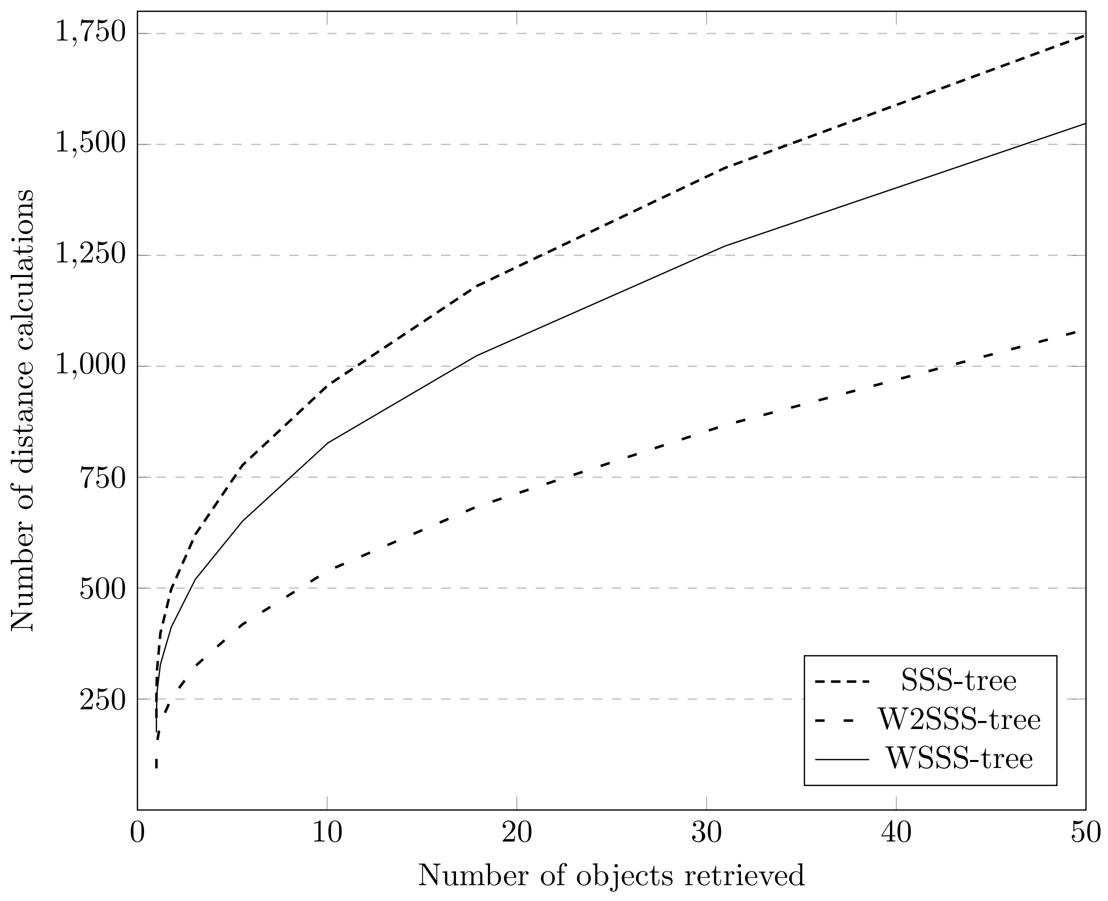


Figure 8.4: The number of distance calculations conducted by the SSS-tree, the W2SSS-tree, and the WSSS-tree, when retrieving 1 to 50 objects by applying queries from two clusters to 100 000 vectors of five dimensions distributed as ten Gaussian clusters.

8.2.5 Unitary Cube of Ten Dimensions

The result of Experiment 3 on 100 000 synthetic random vectors in the unitary cube of ten dimensions is found in Figure 8.5. When retrieving only one object, the W2SSS-tree needs 60% fewer distance calculations than the SSS-tree, and 50% fewer distance calculations than the WSSS-tree. When retrieving closer to 50 objects, the WSSS-tree is more than 30% better than the SSS-tree, and about 20% better than the WSSS-tree. The maximum percentage of the data set reviewed by the W2SSS-tree is slightly more than 10%.

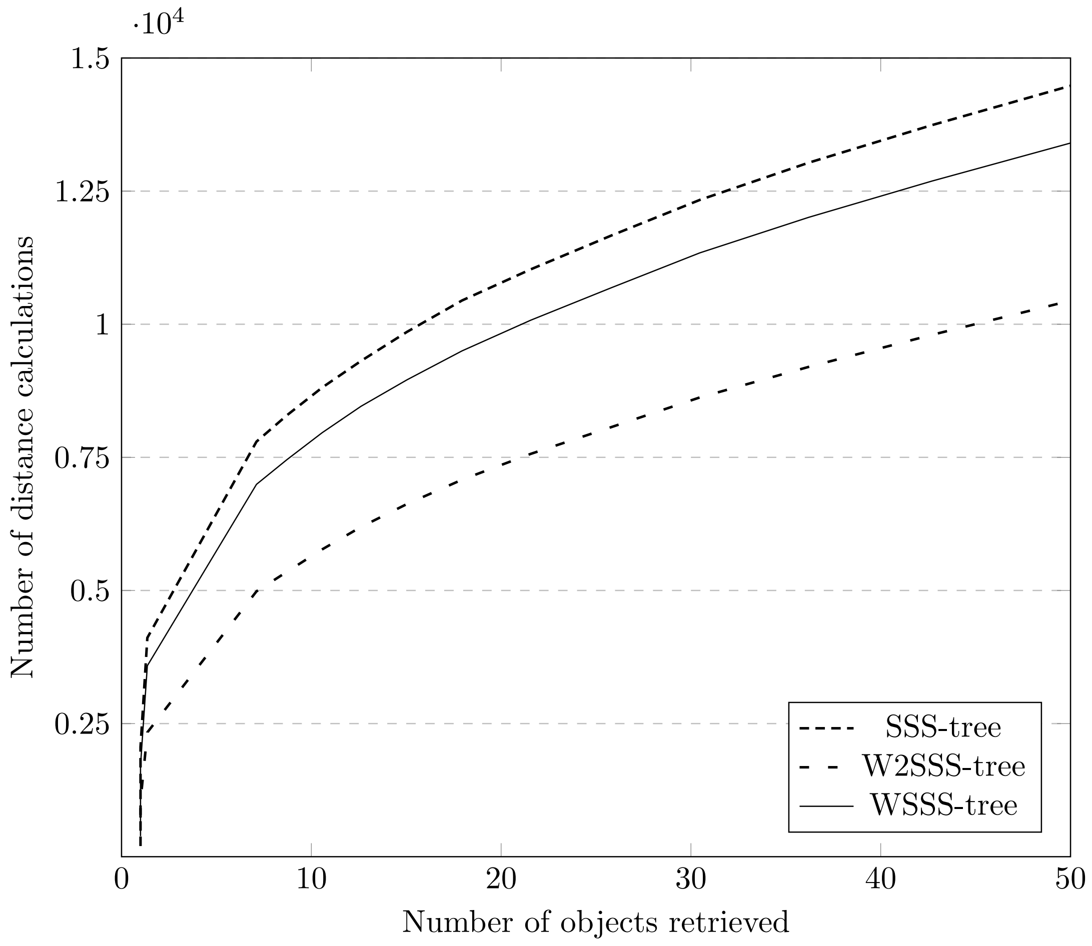


Figure 8.5: The number of distance calculations conducted by the SSS-tree, the W2SSS-tree, and the WSSS-tree, when retrieving 1 to 50 objects by applying queries from two clusters to 100 000 synthetic random vectors in the unitary cube of ten dimensions.

8.2.6 Gaussian Clusters of Ten Dimensions

The result of Experiment 3 on a set of 100 000 vectors of ten dimensions distributed as ten Gaussian clusters is found in Figure 8.6. The W2SSS-tree is 50% more efficient than the SSS-tree, and 35% more efficient than the WSSS-tree, when retrieving only one object. When retrieving up to 50 objects, it performs almost 30% fewer distance calculations compared to the SSS-tree, and 20% fewer distance calculations than the WSSS-tree. The W2SSS-tree reviews about 15% of the data set at most.

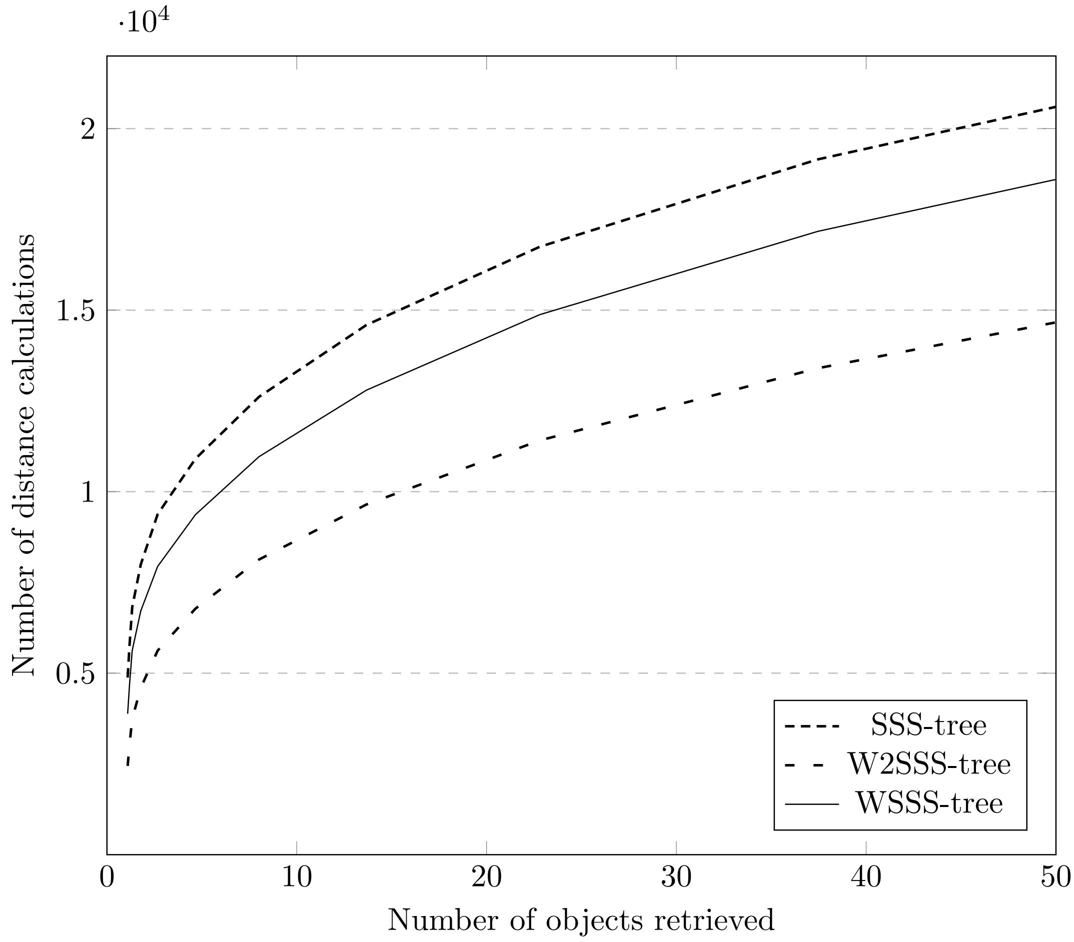


Figure 8.6: The number of distance calculations conducted by the SSS-tree, the W2SSS-tree, and the WSSS-tree, when retrieving 1 to 50 objects by applying queries from two clusters to 100 000 vectors of ten dimensions distributed as ten Gaussian clusters.

8.2.7 Unitary Cube of 15 Dimensions

The result of Experiment 3 on 100 000 synthetic random vectors in the unitary cube of 15 dimensions is found in Figure 8.7. The W2SSS-tree is almost 45% better than the SSS-tree, and almost 35% better than the WSSS-tree, when retrieving one object. When retrieving 50 objects, it is about 10% better than the SSS-tree, and 8% better than the WSSS-tree. At most, the W2SSS-tree reviews slightly less than 50% of the data set.

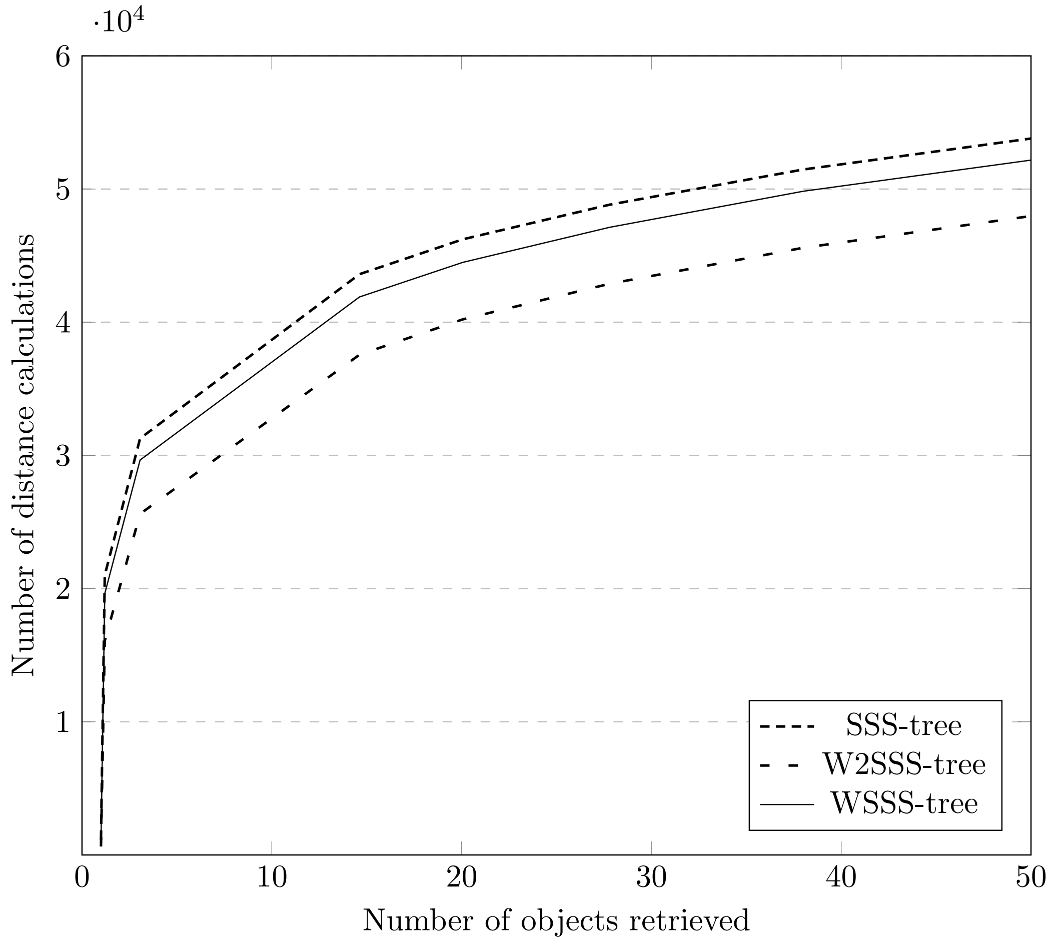


Figure 8.7: The number of distance calculations conducted by the SSS-tree, the W2SSS-tree, and the WSSS-tree, when retrieving 1 to 50 objects by applying queries from two clusters to 100 000 synthetic random vectors in the unitary cube of 15 dimensions.

8.2.8 Gaussian Clusters of 15 Dimensions

The result of Experiment 3 on a set of 100 000 vectors of 15 dimensions distributed as ten Gaussian clusters is found in Figure 8.8. When retrieving only one object, the W2SSS-tree conducts about 30% fewer distance calculations than the SSS-tree and almost 25% fewer distance calculations than the WSSS-tree. When retrieving around 50 objects, the W2SSS-tree conducts almost 15% fewer distance calculations than the SSS-tree, and 10% fewer than the WSSS-tree. The highest percentage of the data set reviewed by the W2SSS-tree is 50%.

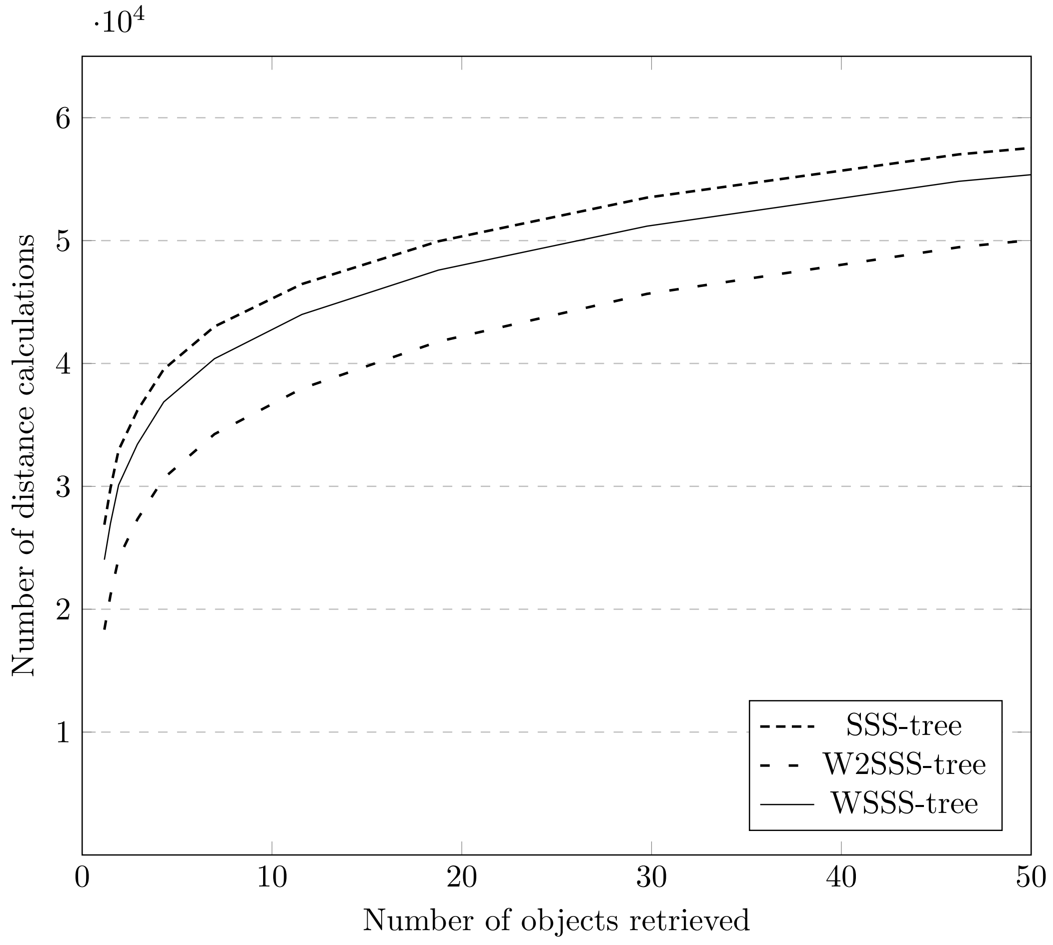


Figure 8.8: The number of distance calculations conducted by the SSS-tree, the W2SSS-tree, and the WSSS-tree, when retrieving 1 to 50 objects by applying queries from two clusters to 100 000 vectors of 15 dimensions distributed as ten Gaussian clusters.

8.3 Summary and Discussion of the Results

The W2SSS-tree performs better than both the SSS-tree and WSSS-tree in all the metric spaces in this experiment. The most significant difference is found when retrieving one object from the vectors from the unitary cube of ten dimensions. In this scenario, the W2SSS-tree computes 50% fewer distances than the WSSS-tree, and 60% fewer than the SSS-tree. This is the metric space where we found the most significant advantage for the weighted trees in the other experiments as well.

The difference in performance is smallest when retrieving 50 objects from the vectors from the unitary cube of 15 dimensions. In this scenario, the W2SSS-tree conducts 8% fewer distance computations than the WSSS-tree and 10% fewer distance computations than the SSS-tree. However, in this space, all indices, including the W2SSS-tree, have to review almost half of the data set when retrieving 50 objects, which suggests that using the W2SSS-tree in this scenario could be inefficacious. All indices also reviewed more than half of the data set in the data set made of Gaussian clusters of 15 dimensions when retrieving 50 objects, and this suggests that the W2SSS-tree, similarly to other weighted indices in this report, is unable to overcome the curse of dimensionality.

Overall, the W2SSS-tree performs significantly better than the SSS-tree and the WSSS-tree. However, we observe that when the dimensionality grows from 10 to 15, the difference in performance drops, just as in Experiment 1. Still, the W2SSS-tree performs noticeably better than the SSS-tree and the WSSS-tree on these data sets as well.

All of these observations support Hypothesis 3.

Chapter 9

Discussion and Concluding Remarks

In this chapter, we will discuss the potential downsides of the weighted indices presented in this report. We will also present a conclusion of the report and suggestions for future work.

9.1 Discussion

The experiments conducted in this report show results in favor of the weighted trees, the WSSS-tree, CSSS-tree, and W2SSS-tree, presented in this report. However, there are several limitations to these trees.

9.1.1 Memory Usage

The weighted trees require more memory than their corresponding SSS-trees. In an SSS-tree, one only has to store one radius for each region. For each region in a WSSS-tree, one has to store the same number of weights as the node has siblings, in addition to the radius. This means that for each node, one will need to store as many weights as its number of children squared. The CSSS-tree uses the same space as the WSSS-tree plus one extra radius for each region. The W2SSS-tree will need to store twice the amount of weights as the WSSS-tree.

When creating indices, one has to consider a trade-off between the amount of memory the index uses, and the query speedup it can provide. This means that to make a fair performance comparison between two indices, they have to take up the same amount of space in memory [11]. Therefore, for the comparisons in this report to be fair, one would have to make modifications to the weighted trees or allow the SSS-tree to include more clusters so that the size of all indices in memory would be equal.

9.1.2 Build Cost

As the weighted trees are more complex than the SSS-tree, the cost of building these trees exceeds the building cost of the SSS-tree. In addition to the cost of using a linear program to optimize the weights of the tree, multiple additional distance calculations are required for all non-leaf nodes to calculate the parameters used in the linear program.

For the reduction in the number of distance calculations performed when searching a WSSS-tree to outweigh the cost of building it, the building cost has to be amortized over a series of queries [15].

9.1.3 Query Distribution

The weighted trees use training queries to optimize their regions. These training queries are selected based on the assumption that they are representative of the queries the index will be used on. More specifically, their average distance to the foci will have to be representative of the queries that it will be applied to. If one, for example, knows nothing of where the queries will come from, this assumption will not be satisfied, and one can not expect the weighted trees to provide a query speedup. In such a case, the weighted trees will be less efficient than the SSS-tree in space without providing a query speedup, and, therefore, not be competitive indexing methods.

9.1.4 Scalability in High Dimensions

In the experiments that the weighted trees had the most significant advantage, experiments 1 and 3, we observed that this advantage dropped when going from data sets of 10 dimensions to data sets of 15 dimensions.

This could be explained by the curse of dimensionality, which refers to how high dimensions can cause data to appear equidistant from each other, making it harder to create efficient indices [6].

A significant open problem in metric space searching is to find efficient solutions for searching high-dimensional metric spaces [6]. The results of our experiments suggest that the proposed weighted indexing methods can not solve this problem, as its advantage starts decreasing already at a dimension of 15.

9.2 Conclusion

We have conducted experiments using three clustering-based metric indexing methods, all of which are based on modifying the SSS-tree to have multi-focal regions optimized by a set of training queries. The experimental results showed the number of distance calculations the indices used to perform exact searches retrieving 1 to 50 objects on vector spaces of dimension 5, 10 and 15, a collection of strings and a collection of images, with queries from one and two clusters.

The WSSS-tree performed significantly better than the SSS-tree when searching for queries from one cluster, and slightly better when searching for queries from two clusters. These results support hypotheses 1 and 2. The CSSS-tree performed slightly better than the WSSS-tree, for both query distributions. When searching for queries from two clusters, the W2SSS-tree performed significantly better than both the SSS-tree and WSSS-tree, supporting Hypthesis 3. This indicates that all the proposed weighted trees can be competitive indexing methods in terms of efficiently conducting queries when disregarding memory usage. However, this is on the condition that one knows the distribution of the queries the indices will be applied to and that one can find a set of training queries that are representative of this distribution. The results of the experiments conducted in this report suggest that this is the case if the queries come from one cluster and are applied to the WSSS-tree, and if the queries come from two clusters and are applied to the W2SSS-tree.

9.3 Future Work

In this report, we have conducted experiments whose results suggest that weighted versions of the SSS-tree can perform better than the SSS-tree when queries come from one and two clusters. There is reason to believe that when the number of clusters the queries come from is higher than two, a weighted tree can still perform better than an SSS-tree if the tree has as many facets as there are query clusters. Further research needs to be done on trees with more than two facets.

There is reason to believe that there are other scenarios than those presented in this report where weighted multi-focal index structures can outperform existing methods. Further investigation is needed to find other query distributions and metric spaces where weighted tree structures will have a significant advantage.

As the memory usage of the weighted trees in this report is higher than the memory usage of the SSS-tree, the comparisons in this report are not fair. It is worth looking into making modified versions of the indices so that they all have equal memory usage.

The optimization of the weights in the trees in this report is based on the average distance from a region's foci to a set of training queries. Further research can be done to find other ways of representing the training queries than just the average distance. This could result in a more accurate representation of the training queries and might improve the results of the weighted trees. It could even remove the need to use as many facets as there are query clusters to get the same result.

Further research can also be done on distinguishing types of queries from one another. If there is a pattern in how large the queries radii are and where they are placed, one could find a way to avoid the different areas of queries with different margins based on this.

In this report, we used modified versions of the SSS-tree. Further research can be done to see if other existing clustering-based index structures are suited to use with weighted multi-focal ambit regions.

Acknowledgements

We wish to thank our supervisor, Magnus Lie Hetland, for the idea behind the experiments conducted in this report, as well as guidance and support throughout this project. All figures of ambits in this report are created using Hetland's implementation for drawing ambits.

Bibliography

[1] Julia 1.2 documentation. `https://docs.julialang.org/en/v1/`.

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

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

[4] Sergey Brin. Near neighbor search in large metric spaces. In *Proceedings of the 21st VLDB Conference*, pages 574–584, 1995.

[5] Nieves Brisaboa, Oscar Pedreira, Diego Seco, Roberto Solar, and Roberto Uribe. Clustering-based similarity search in metric spaces with sparse spatial centers. In *SOFSEM 2008: Theory and Practice of Computer Science*, pages 186–197, 2008.

[6] Edgar Chávez and Gonzalo Navarro. A compact space decomposition for effective metric indexing. *Pattern Recognition Letters*, 26(9), 2005.

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

[8] Lu Chen, Yunjun Gao, Baihua Zheng, Christian S Jensen, Hanyu Yang, and Keyu Yang. Pivot-based metric indexing. In *Proceedings of the VLDB Endowment*, volume 10, pages 1058–1069, 2017.

[9] Ankita Chokniwal and Manoj Singh. Faster mahalanobis k-means clustering for gaussian distributions. In *2016 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pages 947–952, 2016.

[10] Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Proceedings of the 23rd VLDB conference*, pages 426–435, 1997.

[11] Ole Edsberg and Magnus Lie Hetland. Indexing inexact proximity search with distance regression in pivot space. In *Proceedings of the Third International Conference on SImilarity Search and APplications*, pages 51–58, 2010.

[12] Karina Figueroa, Edgar Chávez, Gonzalo Navarro, and Rodrigo Paredes. On the least cost for proximity searching in metric spaces. In *International Workshop on Experimental and Efficient Algorithms*, pages 279–290, 2006.

[13] Aristides Gionis, Piotr Indyk, Rajeev Motwani, et al. Similarity search in high dimensions via hashing. In *Proceedings of the 25th VLDB Conference*, volume 99, pages 518–529, 1999.

[14] Sariel Har-Peled. *Geometric approximation algorithms*. Number 173. American Mathematical Soc., 2011.

[15] Magnus Lie Hetland. *The Basic Principles of Metric Indexing*, pages 199–232. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[16] Magnus Lie Hetland. Ptolemaic indexing. *CoRR*, abs/0911.4384, 2009.

[17] Magnus Lie Hetland. Comparison-based indexing from first principles. *arXiv preprint arXiv:1908.06318*, 2019.

[18] Seth Hettich and Stephen D Bay. *The UCI KDD Archive: Corel Image Feature [http://kdd.ics.uci.edu/databases/CorelFeatures/CorelFeatures.html]*. University of California, Department of Information and Computer Science, 1999.

[19] Tim Holy. Revise.jl. `https://github.com/timholy/Revise.jl`, 2020.

[20] JuliaIO. Bson.jl. `https://github.com/JuliaIO/BSON.jl`, 2020.

[21] JuliaOpt. Clp.jl. `https://github.com/JuliaOpt/Clp.jl`, 2020.

[22] JuliaOpt. Jump.jl. `https://github.com/JuliaOpt/JuMP.jl`, 2020.

[23] JuliaStats. Distances.jl. `https://github.com/JuliaStats/Distances.jl`, 2020.

[24] JuliaStats. Distributions.jl. `https://github.com/JuliaStats/Distributions.jl`, 2020.

[25] María Luisa Micó, José Oncina, and Enrique Vidal. A new version of the nearest-neighbour approximating and eliminating search algorithm (aesa) with linear preprocessing time and memory requirements. *Pattern Recognition Letters*, 15(1), 1994.

[26] Barbara Spillmann. Description of the distance matrices. Institute of Computer Science and Applied Mathematics, University of Bern, 2004.

[27] Shamik Sural, Gang Qian, and Sakti Pramanik. Segmentation and histogram generation using the hsv color space for image retrieval. In *Proceedings. International Conference on Image Processing*, volume 2, pages 589–592, 2002.

[28] Roberto Uribe, Gonzalo Navarro, Ricardo J Barrientos, and Mauricio Marín. An index data structure for searching in metric space databases. In *International Conference on Computational Science*, pages 611–617, 2006.

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

[30] Pavel Zezula, Michal Batko, and Vlastislav Dohnal. *Indexing Metric Spaces*, pages 1451–1454. Springer US, Boston, MA, 2009.