NTNU
Norwegian University of
Science and Technology

Metric indexing by database techniques

Stian Erlandsen

Master of Science in Computer Science
Submission date: February 2011
Supervisor: Svein Erik Bratsberg, IDI

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

Problem Description

Metric indexing and search is a recent technique to do similarity search. There is a large potential for improved performance by combining algorithms from the metric indexing field with well known techniques from the database field.
There has been built a prototype to implement the LAESA algorithm using traditional database indexes in NTNUstore: B+-trees and R*-trees.

The task of this master thesis is to design and implement a version of the LAESA algorithm which is faithful to the original algorithm. This new LAESA version should be compared with the version that has been implemented already. One of the issues is whether B+-trees or heap files should be basis for the new version. The data sets to be used is the NASA picture data and the COLORS data, both from the SISAP library.

Assignment given: 30. August 2010
Supervisor: Svein Erik Bratsberg, IDI

Abstract

Similarity search is very useful in many applications. Because of the complexity and expensive nature of such search operations, many existing methods require special access methods and cannot be directly integrated with commercial DBMSs(Database Management Systems). NTNUStore is a framework to aid research in this field and focuses on disk-based metric indexing to keep it compatible with commercial DBMSs.

This project has implemented and experimented with a version of the LAESA (Linear Approximating and Eliminating Search Algorithm) in NTNUStore. The results are close to what was represented in Erik Bagge Ottesen's master thesis [10]. The biggest contribution in this project is that the new algorithm solves KNN search without the need of providing a range parameter in the query.

Preface

This thesis is the final work for the Master of Science education at the Norwegian University of Science and Technology. The work is a contribution to the NTNUStore platform which was built by Svein Erik Bratsberg and Magnus Lie Hetland [3] and then expanded by Erik Bagge Ottesen [10] through his master thesis. NTNUStore is based on NEUStore [17].

A LAESA method already exists in NTNUStore. A new version that is true to the original algorithm [8] will be implemented and the two versions will be compared with each other.

Big thanks to my supervisor Svein Erik Bratsberg for the valuable inputs on the research and the writing of this report.

Contents

List of Figures

Acronyms

AESA Approximating and Eliminating Search Algorithm

BS-tree Bisector-tree

DBMS Database Management System

GH-tree Generalized Hyperplane tree

GNAT Geometric Near-neighbour Access Tree

KNN K Nearest Neighbour

LAESA Linear Approximating and Eliminating Search Algorithm

MVP-tree Multi-Vantage Point-tree

RAF Random Access File

SAM Spatial Access Method

VP-tree Vantage Point-tree

Terminology

Candidate Refers to an object that may or may not be a part of the result and exact calculations have to be done in order to determine this.

Base-Prototype See *Pivot*.

Euclidean Space Our intuitive understanding of two- or three dimensional space is the Euclidean space - the principles of Euclidean geometry apply.

Foci See *Pivot*.

Hyperplane In the Euclidean space the hyperplane(midset) will be the set of all points that are equidistant from pivot A and B.

Intrinsic Dimensionality Dimensionality Estimation.

Lower Bound An Objects lower bound defines the lowest possible distance it can be from another object(usually a pivot).

Metric Ball A partitioning method of the metric space.

Metric Space Our intuitive understanding of space is the one-, two- and three-dimensional Euclidean Space. The metric space is a generalization of the n-dimensional space where some mathematic properties must hold true.

Pivot An object in the metric space chosen as a reference point for other objects in the space. Sometimes referred to as *foci* or *base-prototype*.

Pivoting A partitioning method of the metric space.

Prototype Objects in the data-set are sometimes refered to as prototypes.

Similarity Search Finding closest points in a metric space by utilizing a distance function(metric).

Chapter 1

Introduction

Similarity search is a query-by-example search where the result is one(nearest neighbour) or more items(K nearest neighbours) that are similar to the example. What defines two items as similar can vary between applications. In a *geometric* model, the distance between the items determine how similar they are. In a *transformational* model however the transformation cost in relation to transforming one item into the other determine how similar they are. The two scenarios below gives a couple examples of similarity search applications.

Scenario 1 *Kate is sitting on the bus on her way home from work. The radio plays a song she has never heard but she really likes it and wants to buy it online. She holds her phone close to the speaker and her application searches for the song by "listening" to it. The application finds out what song it is and returns all the information she needs about the song.*

Scenario 2 *Jonas works as a technical specialist in a law enforcement agency. He receives some video footage taken from a surveilance camera at a crime scene. The footage contains images of the prime suspect. He runs the images against their criminal database and the search engine retrieves the most similar person.*

Similarity search and nearest neighbour problems are ubiquitios, meaning that a solution for one isolated problem might perform well on its own but interoperability is not easily achieved. This is the reason why similarity search is not supported by commercial DBMSs yet. Generic solutions are being researched and improved over time, so one can almost be certain that one day this will change.

A *metric space* is a generalization of space where the metric is a distance function that defines how similar or dissimilar two objects are.

1.1 Metric Space

The formal definition of a metric space is as follows: Let X be a set. A function $d : X \times X \to \mathbb{R}_+$ is called a metric if it satisfies these three conditions:

1. $\rho(x, y) = 0 \iff x = y$ (positive definitness).
2. $\rho(x, y) = \rho(y, x)$ (symmetry).
3. $\rho(x, z) \leq \rho(x, y) + \rho(y, z)$ (triangle inequality).

A **metric space** is a pair (X, ρ), where ρ is a metric or distance function X.

The simplest example of a metric space is the *Euclidean space*. If this space is a two-dimensional space, the metric is a geometric distance function and defined as $\rho(x, y) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}$

Finding the best suited distance function within a given problem domain is a study of its own and is outside the scope of this report. That said, whatever the distance function might be for a certain space, the general concept is a mathematical expression that calculates the distance between two objects.

1.2 Range- and KNN Search

The two most basic types of similarity search are KNN(K-Nearest Neighbour) search and range search. KNN search answers questions like "What is the closest object to object x and what is the distance between these?" or "What are the five closest objects to object x and what is the distance between x and each five?". Range search answers questions like "How may objects are within range of object x using radius y?". The scenario examples given earlier are KNN search where k = 1 since we only want the closest object and nothing else. In range search we are interested in all objects within a specified distance from the query object. This project focus on KNN search.

The most basic solution for the similarity search problem , is to scan the whole dataset and calculate distances from all objects to the query object and return the object with the shortest distance as result. This requires n amount of distance calculations each time we query and since distance calculations are generally viewed as the most expensive factor when it comes to response time, we need a way to reduce this amount. Chapter 2 will explain some existing methods that achieve this. This report will only be focusing on the method called *pivoting*, since LAESA is based on that concept. Reducing the amount of distance calculations is the main objective of LAESA or similar algorithms and is possible through the following three steps:

1. Selecting x amount of pivots(outskirt objects. Sometimes called base-prototype or foci), calculating distances from each pivot to all objects in the set and storing an index for each pivot. This is a very cumbersome process but is only necesarry to run once for a given dataset unless we are trying to figure out an optimal number of pivots through trial and error.

2. Filtering out objects. With the pre-calculated distances now stored in the indexes, some objects can be eliminated by taking advantage of the triangular inequality. How many objects that can be eliminated in KNN search depends on some factors:

- How many pivots that are used.
- How the testobject is positioned in the metric space with respect to the pivot(s) and the other objects.
- How big K is set to. The more neighbours we want, the less pruning can be done.
- The size of the database.

The triangular inequality is a basic property of metric space and is defined as:

$$d(x, z) \leq (x, y) + d(y, z)$$

In range search, this translates to all objects outside the range d(p,q) - r and d(p,q) + r are eliminated, while objects inside becomes candidates. The more pivots used, more objects are eliminated and the candidate set becomes smaller. Albeit choosing too many pivots will give a negative effect in performance since at some point an extra pivot will exclude very few objects, maybe not even a single one. In this case the cost of managing the extra pivot will simply overcome the gain. See figure 1.1 for a visual representation of the filtering process with range search.

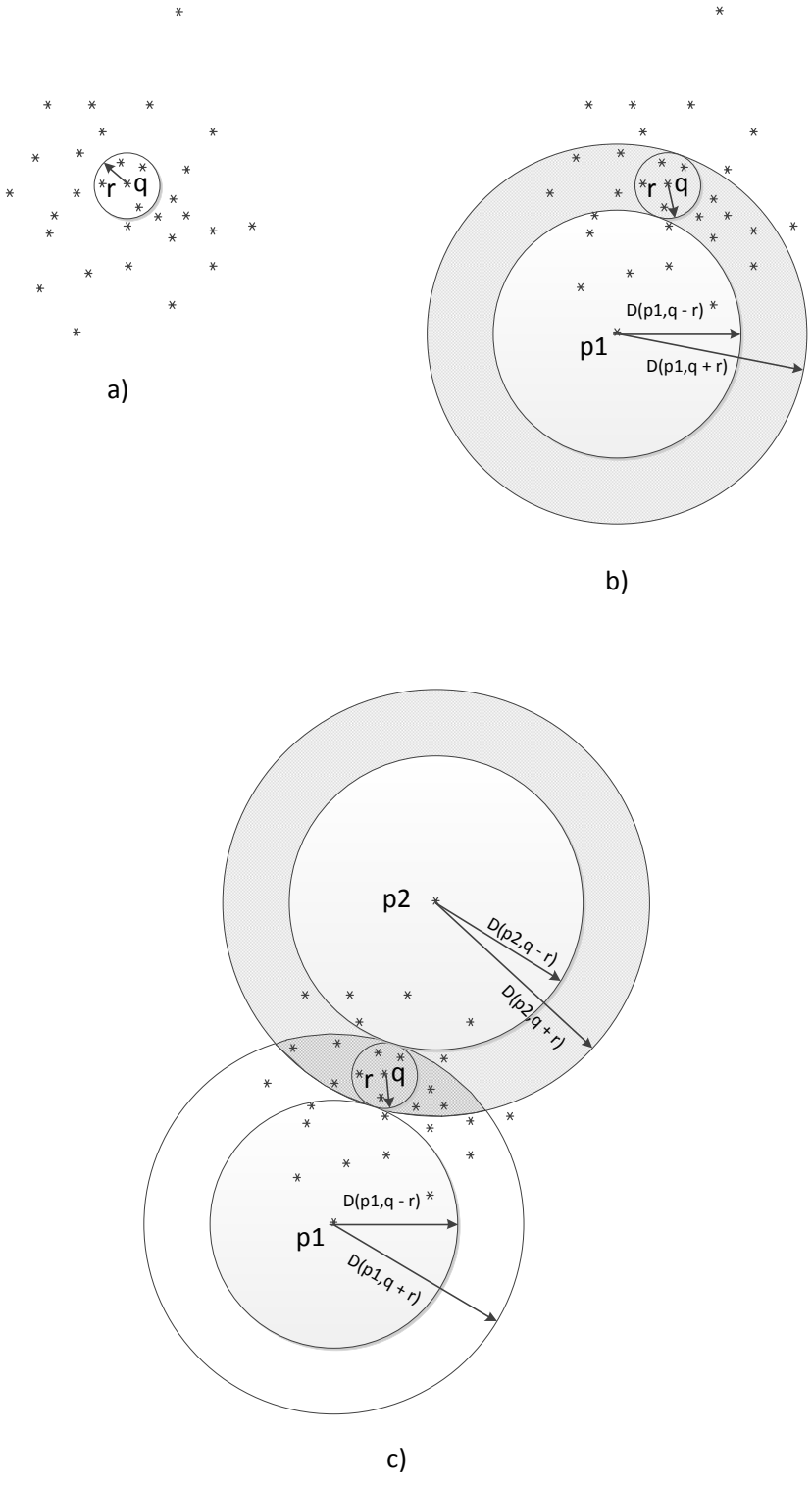


Figure 1.1: a) Set of objects in a two-dimensional space where q is the query object and r is the query range. b) Using one pivot, all objects outside the gray area are eliminated. c) Using two pivots, the gray area becomes smaller and more objects are eliminated compared to b) with one pivot.

This looks a little different with KNN since we do not operate with a range. Instead, the known distances between pivot p and object O_i(stored in the index) and the distance between the query object q and p(calculated by the algorithm) are exploited to create a lower bound for the distance between O_i and q. See figure 1.2

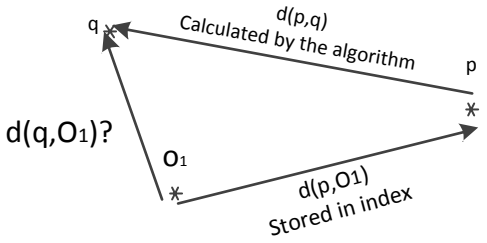


Figure 1.2: How the lower bound is found in KNN search. The lower bound for $\breve{d}(q, O_1)$ is found by $|d(p, q) - d(p, O_1)|$

3. Sort the lower bounds in ascending order. With NN search, stop if the lowest real distance so far is less than the next lower bound. In KNN search, store the lowest K distances found so far and stop if the highest value of these is lower than the next lower bound.

1.3 Motivation, objectives and contributions

1.3.1 General Motivation

The purpose of this thesis is to assist in the NTNUStore project by developing a solution for the original LAESA. A general platform for disk-based similarity search is an important part of the research field and is the architectural choice for NTNUStore. As mentioned earlier, similarity search is at a stage where it is only usable in specialized database systems and not integrated in commercial DBMSs.

1.3.2 Objectives

The goal of NTNUStore is to provide a platform which is able to support multiple similarity search applications. The new LAESA version can serve as a building block towords this goal. NTNUStore, in its current version, is focused towords range search and solves KNN problems through range search. The implementation in this project can provide a new and hopefully better method for solving these query types.

Therefore, the goal of this project is to implement the original LAESA in NTNU-Store and see how it performs compared to existing algorithms.

1.3.3 Contributions

A version of the LAESA has been implemented and presents the following contributions:

- A new algorithm for choosing pivots.
- The LAESA containing four components:
  - NN search using zero pivots.
  - KNN search using zero pivots
  - NN search using one or more pivots.
  - KNN search using one or more pivots.

These components will be explained in more detail in the implementation, chapter 3.

1.4 Challenges and problems

The biggest challenge in this project has been to get a full overview of all the existing classes and methods. Particularly the process of creating, writing and reading indexes involves many classes and single methods tends to jump over several classes before terminating. This makes it challenging to debug problems that might occur with these processes.

The original LAESA presumes that all input information needed is available in main memory. LAESA takes three variables as input:

1. Set of Pivot-object distances. This will take up most of the needed storage space.
2. Set of all objects.
3. Set of all pivots, only keys needed.

This fact does not handshake with the NTNUStore intention, which is to store the data in general access structures on disk to avoid limitations on the dataset size. To not put a constraint on the data-set size, it was decided to store pivot-object distances and all objects on disk while pivot information is stored as random-access files. [enter more here]

1.5 Report outline

The remainder of this report is organized into the following chapters:

Chapter 2 is a survey of related work conducted over the years. It explains the three different methods of divinding the metric plane(*pivoting*, *metirc balls* and *hyperplanes*) and some important work in each of these is given.

Chapter 3 introduces some analysis and descision that had to be taken before and during the implementation. Secondly it explains the two algorithms implemented and how the four search methods were done.

Chapter 4 presents the results from the experiments. The chapter is split between experiments on the NASA data-set and the Colors data-set.

Chapter 5 concludes the project and discusses some challanges that might be interesting for future projects.

Chapter 2

Related work

Metric indexing techniques can be divided into three groups: Pivot-based, metric balls/shells-based and metric planes and Dirichlet domains-based methods. Several methods that provide relatively fast similarity search exists within these groups, albeit very few can be implemented directly into standard DBMSs. As mentioned earlier, this is because they require the set of objects and the index to be kept in memory at run-time. This effectively puts a constraint on the database size.

2.1 Pivot-based methods

Pivot-based methods was introduced in the previous chapter. This chapter is a survey of pivot-based methods and work that has been done in this area.

2.1.1 AESA and LAESA

One might say that the AESA (Approximating and Eliminating Search Algorithm) is the predecessor of other pivot-based similarity search algorithms. It was presented by E. Vidal in 1986 [11] and has been the best known algorithm for a long time. One downside of the algorithm was that it needed to store distances from all objects to all other objects at runtime, that is $O(n^2)$ memory consumption and preprocessing time. Needless to say, the algorithm was only feasible for databases that fit in main memory.

The Linear AESA (LAESA) was presented in [8] and has a time and space complexity that grows linearly with the number of base prototypes(pivots) used. LAESA achieves this through two separate algorithms. The first one selects a small subset of the prototypes(objects), they call this subset for base prototypes. While this selection is going on, the distances between the base prototypes and the other objects are kept in memory for future use. The pseudo code for this algorithm is

shown in figure 3.1.

The second algorithm is a best-first Branch and Bound implementation like the AESA but unlike the AESA, only a subset of interprotoype distances is available. In terms of performance, AESA executes fewer distance calculations than LAESA and is thus still the best choice for small databases. LAESA is not far behind and that is quite impressive considering the circumstances. The pseudo code for this algorithm is shown in figure 3.2.

2.1.2 NTNUStore

NTNUStore is the foundation for this master thesis and is based on NEUStore [17]. It was created by Svein Erik Bratsberg and Magnus Lie Hetland [3] and expanded by Erik Bagge Ottesen [10]. The purpose of NTNUStore is to explore the use of standard database indexes and query processing as a basis for metric indexing. The implementation focuses on range-based search and KNN problems are also solved via range-search. The user have two options when providing a radius for a KNN search. It can either be provided through trial and error or by the *intrinsic dimensionality*. After pivot selection is complete, a *filtering step* prunes objects deemed to be either outside the radius or too far away to be a nearest neighbour depending on which type of query is conducted. The logic for filtering these two query types are almost the same, the main difference is what happens after the candidate sets are joined together. Figure 2.1 and 2.2 shows a filtering example with range search and KNN search respectively. The data-set contains 6 objects and 3 pivots are used. The radius r is set to 1.1 and 1.2 respectively. Note that the numbers used in these examples might be unrealistic and even mathematically incorrect, the only point of the figures is to show the mechanics of the filtering procedure.

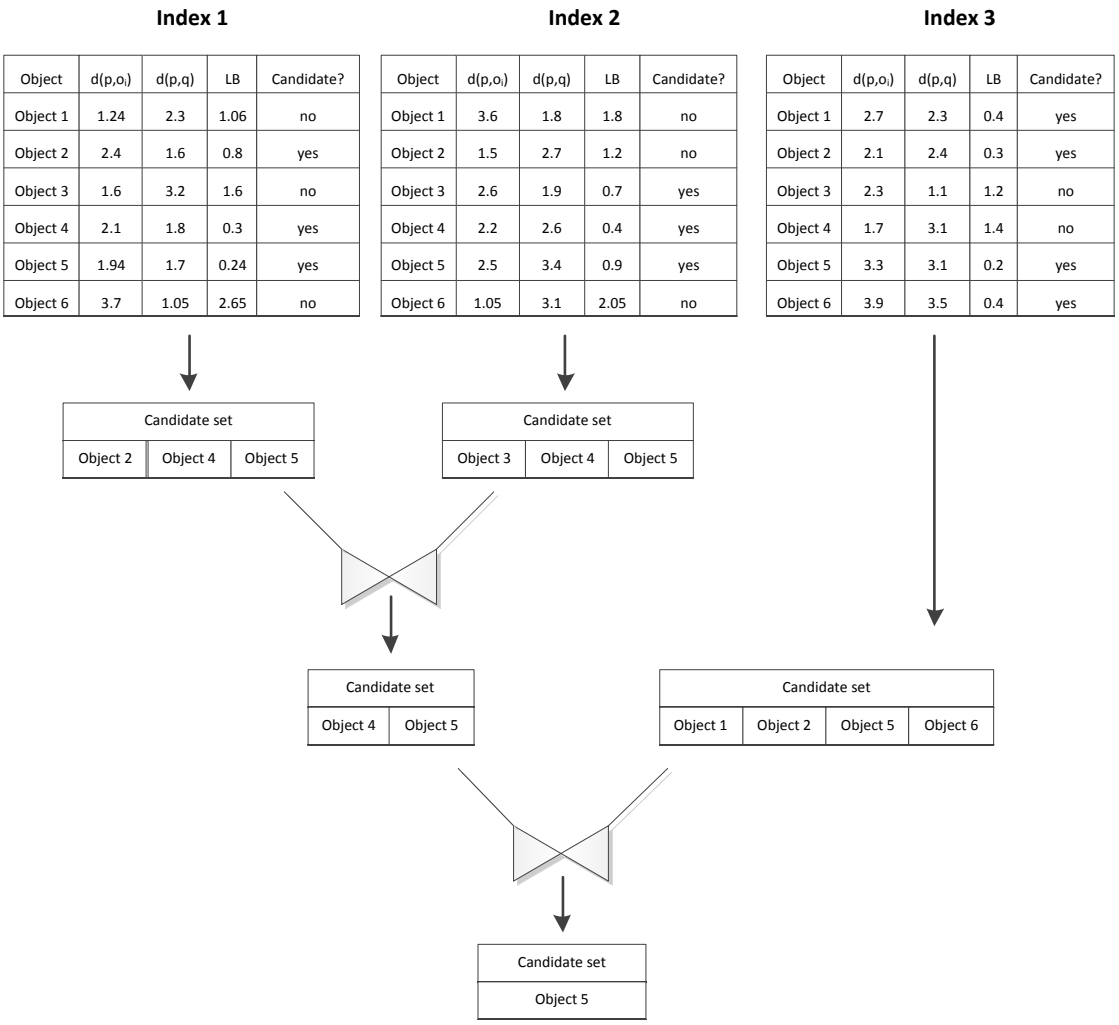


Figure 2.1: Filtering step for range search in NTNUStore with radius r = 1.1.

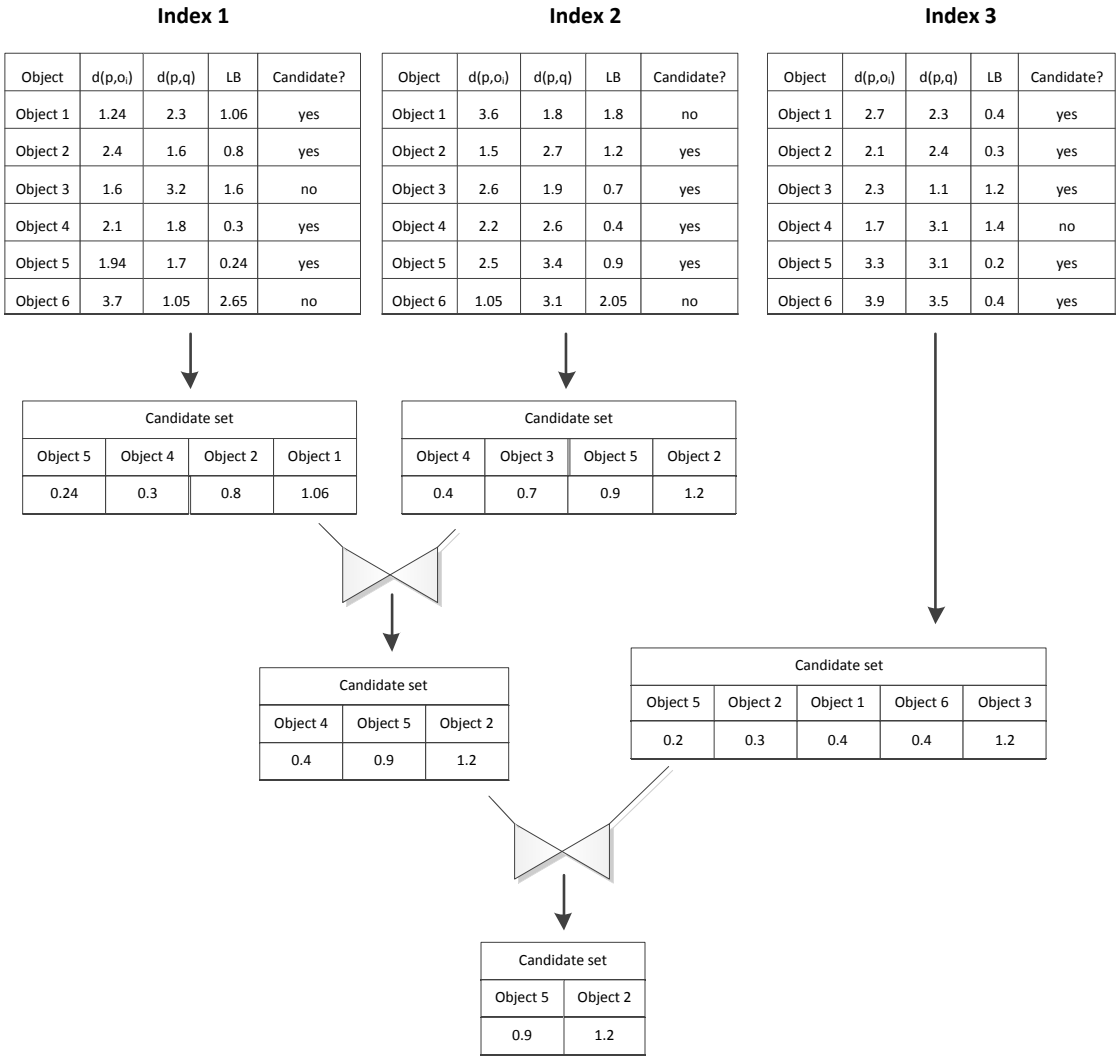


Figure 2.2: Filtering step for KNN search in NTNUStore with radius r = 1.2.

The next step is the *post processing step*. If the query is a range search, the input will be a candidate set returned from the filtering step. This set is formed by intersecting all the sets returned from each index. In other words if a candidate is returned by one index but is not present in every other set, it will be discarded. All exact distances has to be calculated for candidates present after the filtering step.

For KNN search the lower bound distances returned from each index are joined by an equi-join to form the greatest lower bound. One fact that can sometimes be difficult to get around is why one want to keep the highest lower bound values. Why call it lower bound and then we are interested in the highest value? It does indeed require some opposite thinking. Lets say we have three objects A, B and C. We want to find out if B or C is closest to A. Lets say we calculated B to

be 1.2 distance away from A and two pivots gave lower bound values for C to be 0.6 and 1.3. The 1.3 lower bound states that C cannot be closer than 1.3 distance to A and with this information we can discard C without calculating the exact distance. This is why the highest lower bound values are favored. The lower bounds are sorted in ascending order and as each distance calculation is performed, the candidate is moved to an ordered list of K entries. If the next lower bound has a value higher than the K entries found so far, the process stops since the K entries contains the answer.

The access methods developed in NTNUStore are B-tree and R-tree. The test results showed that R-tree is superior in its performance. It is more CPU intensive but can easily be processed on multiple CPUs. Furthermore it was shown that the optimal number of pivots to be used is dependent on the distribution of the data-set and on the query.

2.1.3 Omni-family

The authors of the omni-technique [13] introduces yet another expression for pivots - the *omni-foci*. Three access structures, also called members of the omni-family, are presented: The Omni-Sequential, OmniR-Tree and OmniB-Forest. These access structures requires specialized implementations although they can be implemented on top of standard access methods. Figure 2.3 shows the design of the OmniB-Forest.

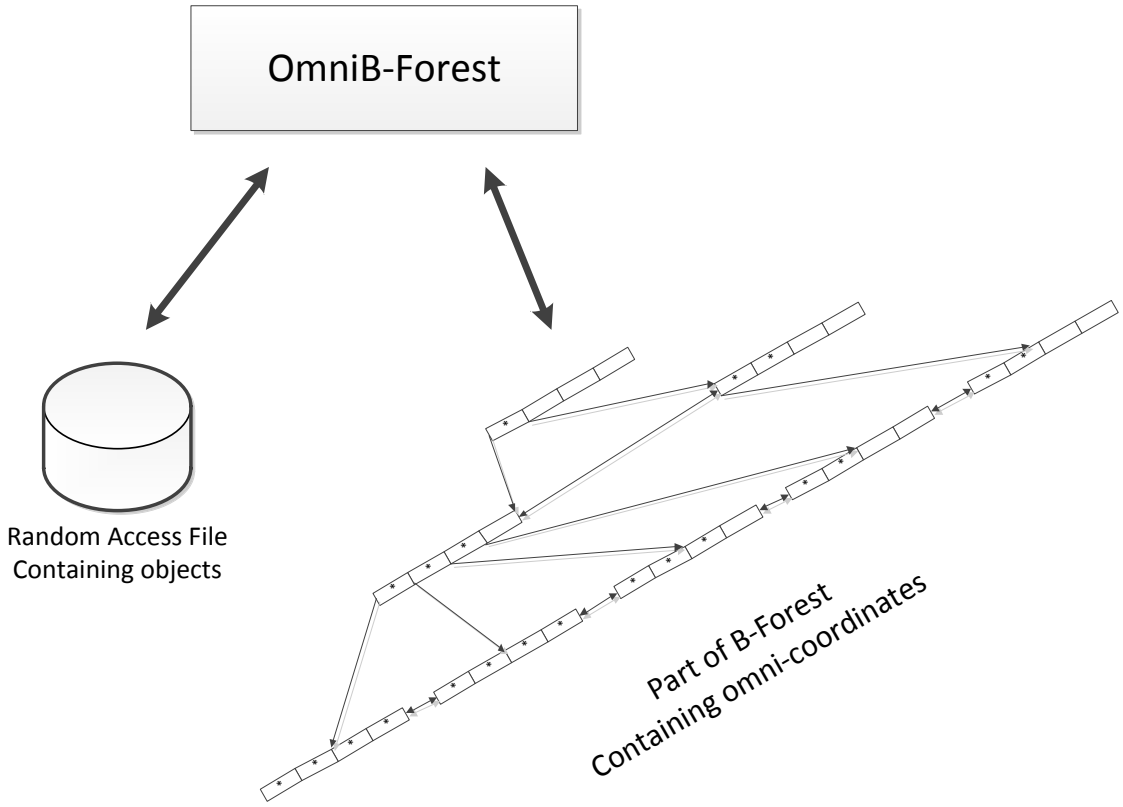


Figure 2.3: Schematic view of the OmniB-Forest.

This design is very similar to NTNUStore. Objects are read from the RAF(Random Access File), focis are selected and distances between each foci and the objects are stored in B^+ -Trees. One B^+ -Tree is generated for each foci and all these trees form the OmniB-Forest.

2.2 Metric ball-based methods

Instead of performing exact distance-calculations to build up an index, Metric balls uses pivots with covering radii that provide an upper bound for the distance between the pivot and any object in the region.

2.2.1 VP-tree, BS-tree and Their Descendants

The *VP-tree* (Vantage Point-tree) [16] [6] uses a single ball to create two regions: One inside and one outside the ball. The object chosen as the first *vantage point*(pivot) becomes the root node in the index and the two regions becomes child nodes of this vantage point, figure 2.4. The radius r is obtained by calculating the

median distance from the vantage point to the remaining objects. This continues recursively and in the next step(figure 2.5) a new vantage point is chosen within S_1's and S_2's bounding regions two form four new nodes. The metric balls will now be smaller since the median distances is now obtained within the subspace. This process continues until all objects have been covered and the resulting index is a static balanced binary tree.

The range query procedure traverses the index from root to leaves. The distance between the pivot of the current node and the query object is evaluated. Lower bounds are required to decide if a subtree needs to be checked or not. In some situations the query object (with its radius) will be covered by both subtrees bounding regions. In such cases both subtrees needs to be checked.

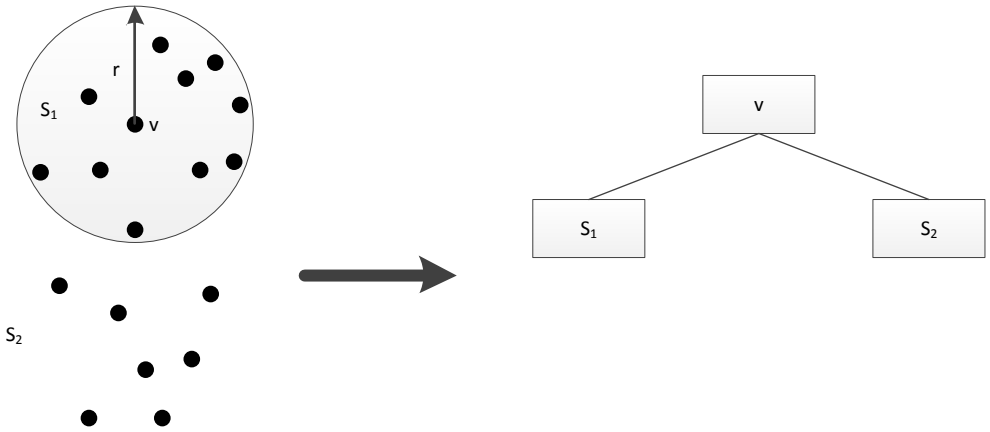


Figure 2.4: Generating the VP-tree, step 1.

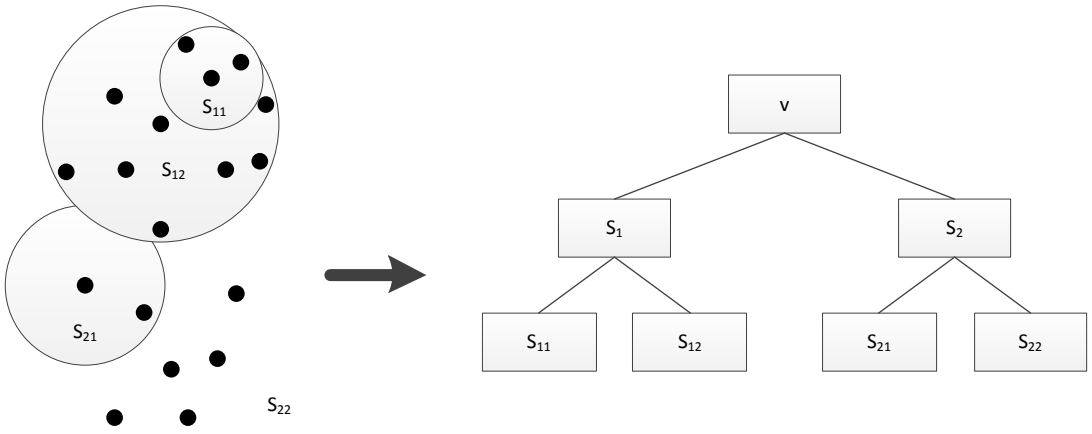


Figure 2.5: Generating the VP-tree, step 2.

The *BS-tree* (Bisector-tree) [12] creates a ball for both regions so the top node has two pivots, P_L for the left subtree and P_R for the right subtree. When an object O_i is to be inserted, it will be inserted into the left subtree if $d(O_i, P_L) < d(O_i, P_R)$

and into the right subtree if $d(O_i, P_L) > d(O_i, P_R)$. If a node does not have two pivots, it will be added as a pivot for this node. The resulting tree is a balanced- or unbalanced dynamic binary tree depending on the distribution of the data-set, see figure 2.6.

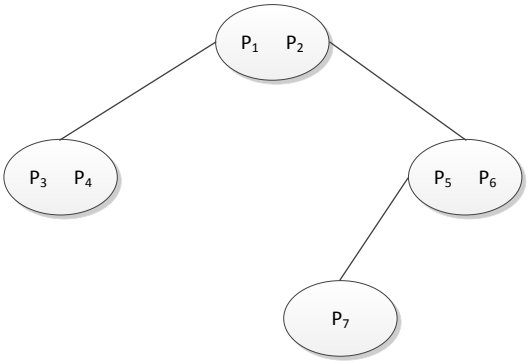


Figure 2.6: Schematic view of the BS-tree.

The *MVP-tree* (multi-vantage point-tree) was proposed by Bozkaya and Ozsoyoglu [1] [2] and is a descendant of the VP-tree. It uses two pivots in each node and multiple radii to partition the dataset into shells.

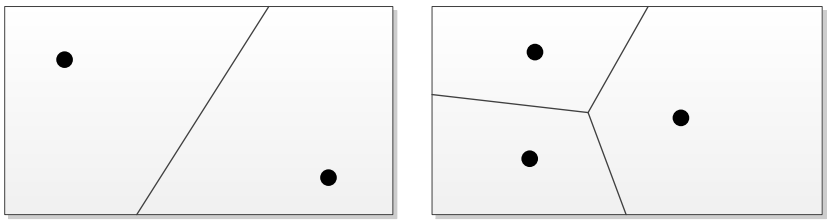
The *M-tree* [5] was an effort to combine the strengths of balanced and dynamic SAMs (Spatial Access Methods). Each subtree is contained in a metric ball and new objects are inserted into the closest ball. The resulting access structure was a dynamic disk based balanced tree. This property can be viewed as the most innovative one and is probably the reason why several later contributions have based their work on the M-tree.

The Slim-Tree [14] is one of these. It uses algorithms that defines and reduces the "fat-factor" of a tree. The fat-factor is a measurement for the overlap between trees where the optimal tree has a fat-factor of 0 and the worst has a fat-factor of 1. Having trees with low fat-factor results in improved query performance. An algorithm called "slim-down" is also presented. This algorithm improves the fat-factor of existing trees.

iDistance [7] focuses on KNN search in high-dimensional metric space. It is also based on metric balls, but what is unique about iDistance is that it transforms the high-dimensional space into a one-dimensional space by a three-step algorithm. The transformed space is indexed in a B^+-tree. This access structure was chosen because it is available in most commercial DBMSs and it is efficient on one-dimensional data.

2.3 Metric planes and Dirichlet domains

The *GH-tree* (Generalized Hyperplane tree)[15] introduces a new(and in some cases perhaps better) way of dividing the space. The metric space can be divided by a hyperplane(midset) between two pivots so that an object is closer to either pivot A or pivot B. In general metric spaces the hyperplane is not defined so easily but in the Euclidean space the hyperplane will be the set of all points in the space that are equidistant from A and B. *GNAT* (Geometric Near-neighbour Access Tree) [4] expands this by using multiple pivots to generate a multi way midset. The resulting regions are called Dirichlet domains. See figure 2.7 for a schematic view of these partitioning styles [6].



(a) (b)

Figure 2.7: a) How the hyperplane is formed in the GH partitioning style. b) How the multi way midset is formed by adding pivot in GNAT partitioning style.

As with pivoting and metric balls, there are several variations to these partitioning styles. A good survey of these can be found in [12] and [6].

Chapter 3

Implementation

3.1 Analysis and Decisions

A couple of decisions had to be made when implementing the LAESA in NTNUStore. One direction was to stay 100% true to the algorithm as described in [8]. Another direction was to use the existing framework in NTNUStore 100%. Fulfilling both was impossible. For example, the NTNUStore framework is based on the idea of using disk based storage to support standard database systems. The LAESA described, on the other hand, focus on storing in main memory. This dilemma and other problems will be elaborated on in this chapter.

3.1.1 The Original LAESA

The method searches for the nearest neighbour(NN-search) of the query object.

The first algorithm selects the pivots while the second algorithm performs the search. Figure 3.1 and figure 3.2 contains the pseudo code for these two algorithms

respectively.

input : $P \subset E$; $m \in \mathbb{N}$; {finite set of objects; number of pivots;}
output: $B \subseteq P$, $|B| = m$; $D \in \mathbb{R}^{|P| \times |B|}$; {set of m pivots; set of distances between pivots and objects;}
Function: $d : E \times E \longrightarrow \mathbb{R}$ {*distance function*}
Variables: $A \in \mathbb{R}^{|P|}$; $b, b' \in P$; $max \in \mathbb{R}$; {*distance accumulator array; used to store pivots; used to store the highest accumulated distance*}

```
begin
    b' ← arbitrary_element(P);
    while |B| < m do
        max ← 0; b ← b';
        for p ∈ P − B do
            D[b, p] ← d(b, p);
            A[p] ← A[p] + D[b, p];
            if A[p] > max then
                b' ← p;max ← A[p];
            end
        end
        B ← B ∪ {b'}
    end
end
```

Figure 3.1: pseudo code for the LAESA pivot selection.

input : $P \subset E$, $n = |P|$; $B \subseteq P$; $D \in \mathbb{R}^{n \times m}$; $x \in E$; {finite set of objects; set of pivots; precomputed $n \times m$ array of distances; test sample;}
output: $p^* \in P$; $d^* \in R$; {nearest neighbour; the distance to NN;}
Functions: $d : E \times E \to \mathbb{R}$ {*distance function*}
Variables: p, q, s, $b \in P$,dxs, gp, gq, $gb \in \mathbb{R}$;
$G \in \mathbb{R}^n$ {*lower bounds array*};
$nc \in \mathbb{N}$ {*number of computed distances*};
$CONDITION$: *Boolean* {*controls the elimination of pivots*};
$CHOICE : B \times (P - B) \to P$ {*selection of object(non-pivot) or pivot*}

```
begin
    d* ← ∞; p* ← indeterminate; G ← [0];
    s ← arbitrary_element(B); nc ← 0;
    while |P| > 0 do
        dxs ← d(x, s); P ← P − {s}; nc ← nc + 1;
        if dxs < d* then
            p* ← s; d* ← dxs;
        end
        q ← indeterminate; gq ← ∞; b ← indeterminate; gb ← ∞;
        for every p ∈ P do
            if s ∈ B then
                G[p] ← max(G[p], |D[p, s] − dxs|);
            end
            gp ← G[p];
            if p ∈ B then
                if gp ≥ d*&CONDITION then
                    P ← P − {p};
                end
                else
                    if gp < gp then
                        gb ← gp; b ← p;
                    end
                end
            end
            else
                if gp ≥ d* then
                    P ← P − {p};
                end
                else
                    if gp < gq then
                        gq ← gp; q ← p
                    end
                end
            end
        end
        s ← CHOICE(b, q);
    end
end
```

Figure 3.2: Pseudo code for the LAESA.

A big while loop covers the entire search algorithm and runs through each object one at a time. If the current object is a pivot, a lower-bound array from this

pivot to all objects is computed. If the current object is a non-pivot, it is either eliminated or the distance is calculated. Two parameters, *choice* and *condition*, allow for different strategies concerning pivot management. *choice*, decides wether the next object sohuld be a pivot or a non-pivot and *condition* decides if the elimination of a pivot can occur or not. The purpose of these two parameters is to add optimizing abilities direcly into the algorithm. These parameters might be redundant if a seperate optimizer is developed.

An attempt to implement the original LAESA in NTNUStore was conductet but with storage on disk and not in main memory(The source code can be found in Appendix A). It managed to calculate NN correctly, but something caused it to run in suspiciously long time. This was caused by either misinterpreted pseudo code or an error in the pseudo code itself. Either way at some point the algorithm looped thousands of times without eliminating objects. This problem was difficult to pinpoint and in the end the decision was made to rewrite the alorithm in a simpler form. The rewritten algorithm is very similar to the one explained in [9] and supports KNN-search in addition to NN-search. The two differences worth mentioning is that the rewrite does not use *condition* nor *choice* and it processes lower bounds for all the pivots first before continuing with non-pivots. The source code of this final version is given in Appendix B.

3.1.2 Existing Framework

The existing framework in NTNUStore is focused on range-queries and so-called *c*ursors that contain results from the range-queries. The methods requires a range input r together with the query object and the existing KNN-search also requires a range input. It would have been easier to compare results with the existing methods if the implemention in this project also used range-based search. Due to time-constraint and being true to the original algorithm, it was decided against this.

With the reservation that nothing was overlooked in the source-code, the existing BTree-framework does not provide an easy solution for extracting pivot-related information such as key and data. A very minimal helper class, *pivot*, was implemented for this purpose. This was much less timeconsuming than rewriting/debugging the quite complex BTree structure.

3.2 Implementation Part 1 - Pivot Selection

The implementation is split into two seperate parts. LaesaPivotSelection is the first one and as you may have guessed, it selects the pivots. The LaesaPivotSelection algorithm is prsented in this section, while the LaesaAlgorithm is covered in section 3.3.

Tools used in the implementation are:

- Eclipse Version: Helios Release, Build id: 20100617-1415
- Java SE Version: 1.6.0_21.

The pivot-selection method is called *laesaPivotSelection* and is an implementation of the method found in [8]. There is one slight difference, the first randomly selected object is not included in the final set of pivots. The laesaPivotSelection first picks a random object $o1$. This object is actually just the first object in the set, similar to [10]. Then it finds the object $o2$ furthest away from o1. The object furthest away from o2 will become o3 and so on. This goes on until the number of pivots provided by the user plus one is reached. The reason for adding one extra, is that the first randomly picked object o1 will be deleted at the end. Reason being, o1 might not be an outer-edge object and thus might not perform well as a pivot.

The laesaPivotSelection builds a B-tree for every pivot in each loop with support from the exisitng LaesaBTree framework. These trees are stored in the ntnustore/trunk/data folder and is given the name *pivot[intkey=x]* where x is the key of the pivot. This is just to visiualize which object has been selected. Two additional random access files are created; pivotKeys and pivots. pivotKeys is just an array containing all the keys of all the pivots and the pivotsfile is a hashmap of all pivots. The hashmap also contains the pivot keys but additionally has a mapping key -> FloatArrayData. These two files would be redundant if it was possible to retrieve this information directly from the B-trees.

3.3 Implementation Part 2 - Search Algorithm

The method takes three maps(O, P and D), one array(PK), one testobject(x), one matrix (only used with QFDistance) and an int(k). Maps were often the most natural choice since the objects have a key - data relationship.

- O: A map containing all the objects in the set. Both pivots and non-pivots. The map maps an IntKey to a FloatArrayData.
- P: A map containing all the pivots selected by the algorithm described in the previous section. The pivot's IntKey maps to the pivots FloatArrayData.
- D: This map contains all the pivots BTrees or indexes and the pivots IntKey points to the respective BTree.
- PK: This is an array containing the keys to all the pivots.
- x: The testobject. This object is randomly selected from the set contained in the map O. When the searching algorithm iterates, it will delete the object x when it stumbles upon it, otherwise the algorithm will give out wrong nearest neighbor with distance 0.0(the distance to itself).

- k: The number of closest neighbors to calculate. This nubmer is provided by the user.

The logic is split into two seperate parts, part A and part B. Part A is chosen when no pivots are provided, when PK.size equals zero. If one or more pivots are provided, part B is chosen. Both these two parts are again split, depending on if the user asked for just the nearest neighbor(k = 1) or several neighbors(k > 1). See figure 3.3 for an overview.

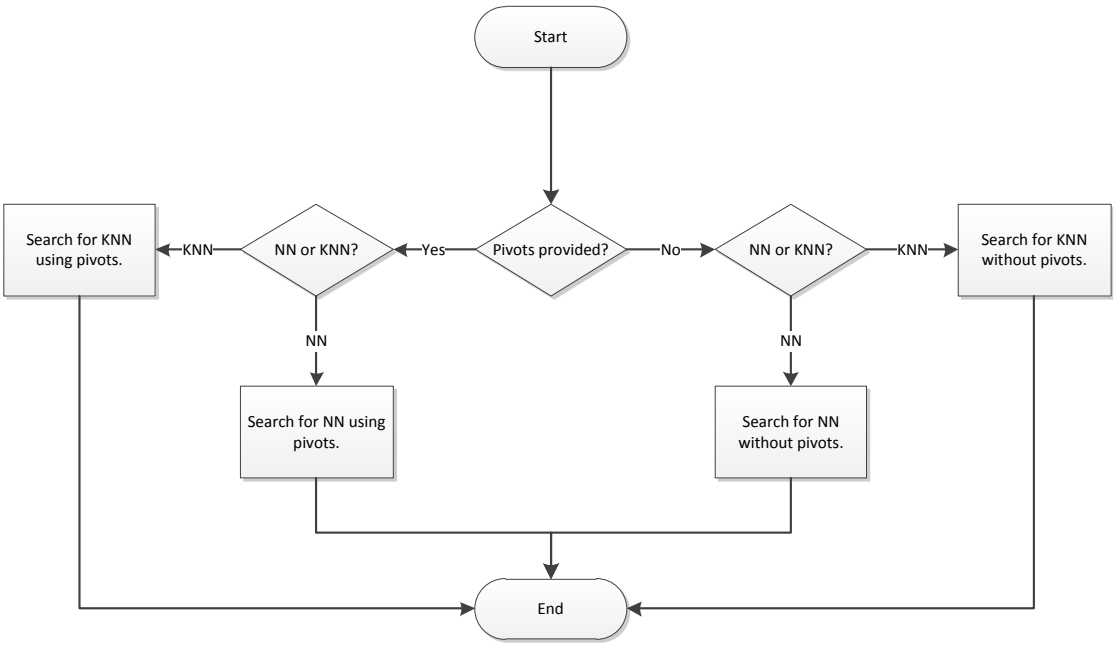


Figure 3.3: Logic for the LAESA algorithm.

3.3.1 NN Search - No Pivots

This is the least complicated problem of the four. One variable *pmin* keeps the lowest distance found so far. Pmin starts out infinately large and while looping through all the objects one at a time, the distance between the current object and the testsubject is stored in the variable *dpx*. Pmin will replace its stored value when dpx contain a smaller distance. After looping through all objects, pmin will contain the lowest distance between the winner and the testsubject. This procedure obviously require n distance calculations. An example of this procedure is shown in figure 3.4

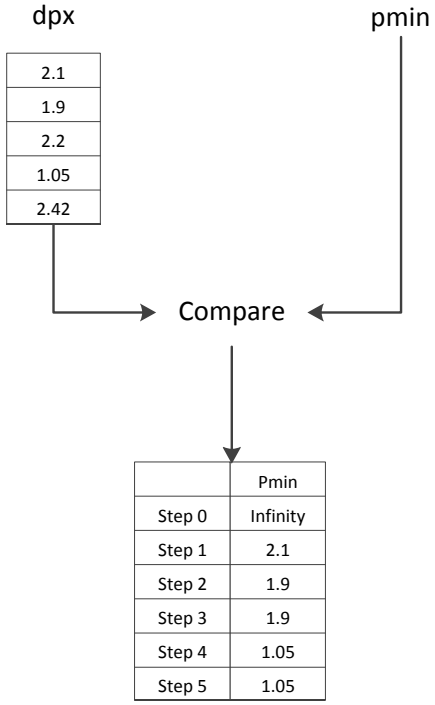


Figure 3.4: An example showing a few steps of the procedure when using no pivots and k = 1.

3.3.2 KNN Search - No Pivots

This procedure is very similar to the previous one. The main difference is that one variable is not good enough for storing the lowest distance since K distances needs to be stored. It also needs to answer the two questions:

- How to compare the current distance with the distances found so far?
- How to replace the looser with the current object if the current object has a smaller distance?

This is done by using a sorted array[k] and two procedures. The first procedure is chosen if the array is not filled up yet. It inputs an infinately large distance in the first index to begin with. This is just for the first object to have a distance to compare against. If the current object has a lower distance than array[0], array[0] is shiftet right and array[0] is replaced by the current object. See figure 3.5 a) for an example of this procedure.

The second procedure takes over when the array is filled with objects. Since the array is sorted, it starts to compare from the right instead of left of the array. When the correct spot for the current object is found, all objects from this spot and to the right is shiftet right. The rightmost object is pushed out of the array

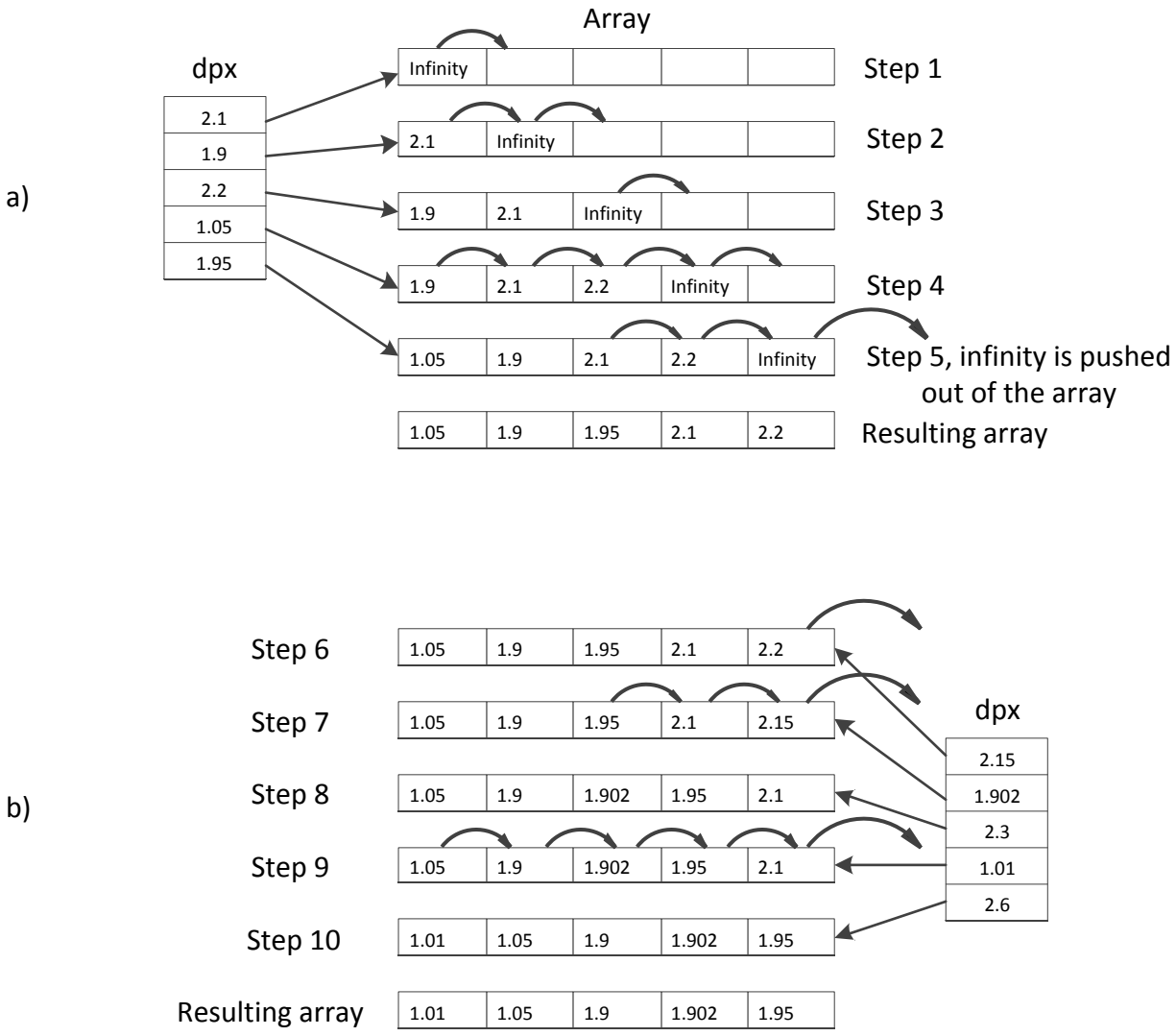


Figure 3.5: An example showing a few steps of the procedure when using no pivots and k = 5. The procedure in a) is run when the array is not filled up (not counting infinity) and compares values from left to right. The procedure in b) takes over when the array is filled up and compares from right to left.

and becomes a looser. Finally, the object is inserted to its rightfull place. After all objects have been calculated, the array will contain the k closest neighbors in sorted order. See figure 3.5 b) for an example.

3.3.3 NN Search - With Pivots

A map containing lower bound values for every object is calculated first. Let x be a testobject, p is a pivot and o is a normal object. The lower bound for the distance between o and x is found by the formula:

$$g[o] = MAX_{i=1}^{n_p}(|d(p_i, o) - d(p_i, x)|) \tag{3.1}$$

This formula is derived from the triangle equality. $d(p_i, o)$ is the distance between pivot i and the object and $d(p_i, x)$ is the distance between pivot i and the testobject.

A variable *pmin* stores the best value found so far and is set to infinity initially. After a complete lower bound map is built, the next step sorts the map based on its values. This is because when the map is sorted, we can compare lower bound values from left to right, or lowest to highest, against the best real distance found so far. When the lower bound value is lower than the best distance found so far, the exact distance needs to be calculated. This exact distance is compared with pmin and pmin will replace its value if this exact distance is lower. If it is higher, the current object can not the nearest neighbor and is discarded. However, when the lower bound value is higher than pmin, we know that the rest of the set will also be higher since the map is sorted. This means that none of the remaining objects can be the nearest neighbor, pmin now contains the answer and the algorithm stops at this point. See figure 3.6 for an example.

Map containing 10 objects with their respective lowerbound values

1.05	1.05	1.26	1.35	1.88	2.223	2.46	2.48	2.55	2.66

	Object i	Lowerbound	Exact distance	< or > ?	Pmin before	Pmin after
Step 1	object 1	1.05	1.99	<	infinity	2.1
Step 2	object 2	1.05	1.89	<	2.1	1.89
Step 3	object 3	1.26	2.3	>	1.89	1.89
Step 4	object 4	1.35	1.95	>	1.89	1.89
Step 5	object 5	1.88	1.45	<	1.89	1.45
Step 6	object 6	2.223				
Step 7	object 7	2.46				
Step 8	object 8	2.48				
Step 9	object 9	2.55				
Step 10	Object 10	2.66				

Figure 3.6: An example showing a few steps of the procedure when using pivots and k = 1. The procedure stops after step 5 since all the following lower bound values are higher than the best real distance found so far(pmin). Object 5 is the nearest neighbor with 1.45 distance to the testobject.

3.3.4 KNN Search - With Pivots

The last procedure handles similarity search when using pivots and k > 1. This procedure is exaclty the same as in previous section 3.3.3 with the exeptions that a) the K best distances so far is sorted in an array, b) the logic that handles the array and whether it is filled up/not filled up is the same as in section 3.3.2 and c) the algorithm does not stop until the array is filled and the current lower bound is higher than the last value in the array.

As one can see in Appendix B, the main method is quite large. This is because it contains all code for handling the tests. The main method performs the following tasks in order:

1. Loops throgh an array that contains the number of pivots that should be tested.
2. Reads the necesarry data from disk that is required by the pivot selection algorithm.
3. Runs the pivot selection algorithm.
4. A variable containing number of testrounds forms an inner loop. If it is set to 100, every number of pivots in the array mentioned above will be tested 100 times each. The pivot selection algorithm is only run once for each number of pivots.
5. Most of the remaining code evolves around gatering the input that is required by the searching algorithm.
6. In the end some statistics are written to textfiles. If a test was run with 5 pivots, there will be a file called *average5.txt* which contains the average number of distance computations for 5 pivots.

Chapter 4

Experiments

The LAESA is aimed for situations where the distance computations are expensive. The existing framework in NTNUStore together with the data-sets (see section for details.) used, computes one distance in almost negligible time. This is unfortunate because in this case it will always be best to run the algorithm without any pivots and this does not depict the real world in a good way. To simulate more costly distance computations, each computation are weighted an extra 1ms. This is exactly the same as was done in [10].

The results shown in each charts are collected by running 100 independent queries and calculating the average. The query object is selected randomly from the whole set of objects in each query and the Euclidean metric is used in all these experiments.

4.1 System Specification

Figure 4.1 contains the system specification used in the experiments. Unfortunately, only a personal computer was available during the experiments so one might see better results on a server system, especially if the server has RAID-0 capability. This is due to the many disk-reads performed in the experiments.

CPU	Intel(R) Core(TM) 2 Duo CPU E6850 3.00Ghz
Memory	4GB main memory
Disk	Western Digital Raptor 1500ADFD
	Rotational speed: 10.000 rpm
	Average Latency 2.99 ms
	Buffer: 16MB
	Average Read Seek Time: 4.6 ms
Operating system	Windows 7 64-bit
Virtual machine	SUN Java SE 1.6.0_21

Figure 4.1: System specification.

4.2 Data Sets

The data sets used in the experiments are two different vector sets; *nasa* and *colors*.

> Nasa is a set of 40,150 20-dimensional feature vectors, generated from images downloaded from NASA (at `http://www.dimacs.rutgers.edu/Challenges/Sixth/software.html`) and with duplicate vectors eliminated. Colors is a set of 112,544 color histograms (112-dimensional vectors) from an image database (at `http://www.dbs.informatik.uni-muenchen.de/~seidl/DATA/histo112.112682.gz`, there are others in the same page) with duplicates removed.

(Figueroa, Navarro & Chávez, 2008).

4.3 Experiments on the Nasa Data-set

This section explains all tests done with the nasa data-set while Section 4.4 deals with all tests concerning the colors data-set.

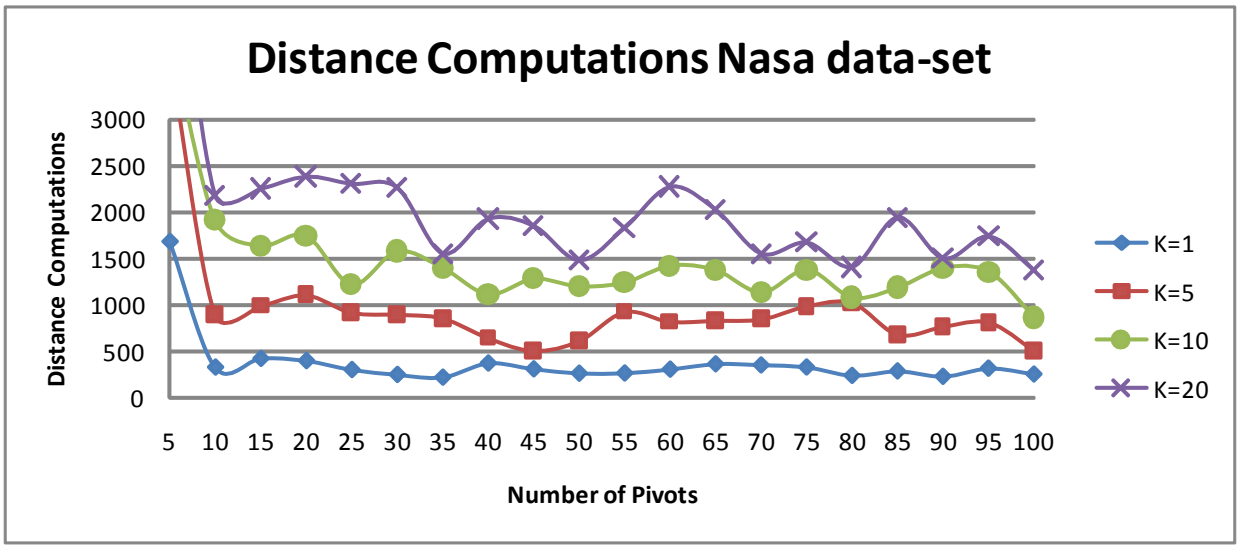


Figure 4.2: Number of distance computations for the nasa data-set with growing number of pivots and with four different K values.

The first obvious observation is that when K grows, the number of distance computations grows. In addition, the increasingly number of pivots has less impact when K is low and bigger impact as K grows. If we observe the curve for K=1, the curve is more or less flat and the number of distance calculations are almost the same with ten pivots versus one hundred pivots.

Although distance computations are considered the dominating factor when it comes to response time, pivot management does also have some impact. The next experiment takes a look at how big this impact is. There are two factors that takes up almost all time when it comes to pivot management; building lower bounds and sorting the lower bounds. This time grows linearly with the number of pivots. Figure 4.3 contains recorded time spent on these two operations with varying numbers of pivots.

Number of pivots	5	10	15	20	25	30	35	40	45	50
Time building lowerbounds (ms)	785	1287	1466	1857	2120	2476	2817	3345	3667	3787

Number of pivots	55	60	65	70	75	80	85	90	95	100
Time building lowerbounds (ms)	4014	4860	5041	6441	7863	9418	8954	11217	10420	12253

Figure 4.3: Time spent on building and sorting lower bounds with varying numbers of pivots.

Lets consider that one distance computation takes 1 ms as an example. Let C_i be the number of distance computations and TLB_i(figure 4.3) be the time it takes to build lower bounds for i pivots. The total response time for i pivots would then approximately be $(C_i \times 1ms) + TLB_i$. Figure 4.4 shows the result of this equation.

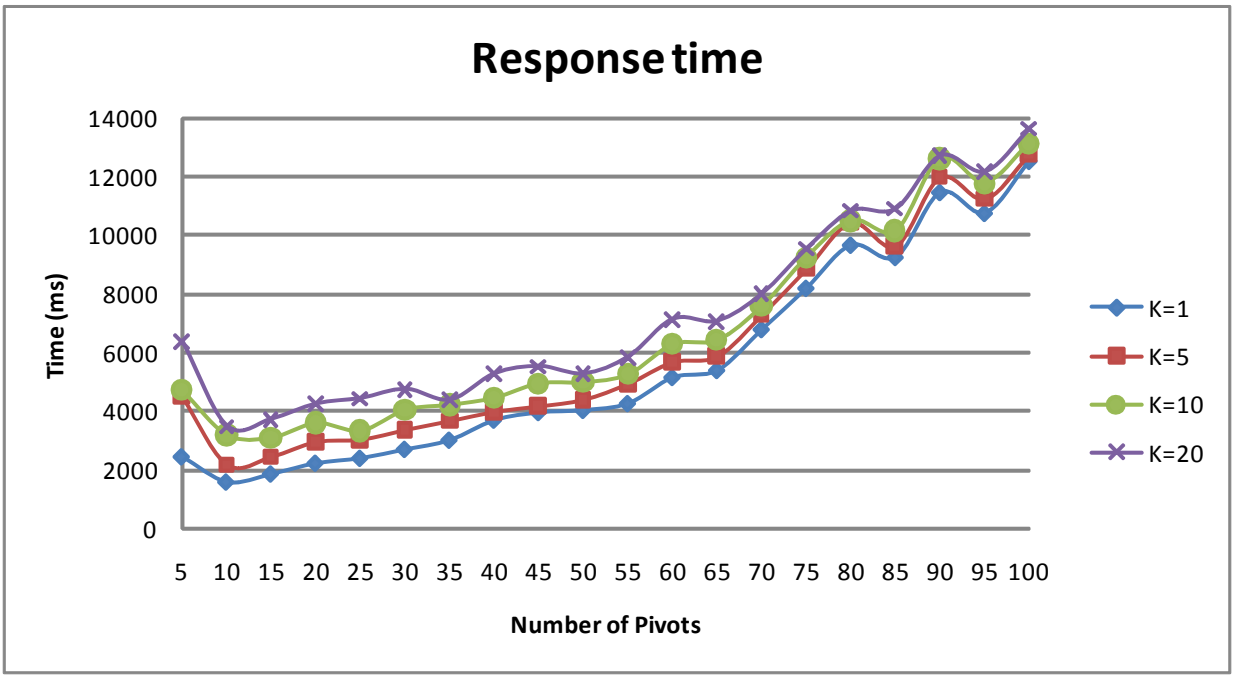


Figure 4.4: Response time for nasa data-set with 1ms distance calculations

The figure above shows that using 10 pivots would be more or less the best choice for all four values of K. This is not unexpected since the number of distance calculations was more or less flat beyond 10 pivots(Figure 4.2). The response time grows in a linear fashion because of time spent on managing lower bounds.

4.4 Experiments on the Colors Data-set

The experiments conducted in the previous section was also performed on the colors data-set and as mentioned, colors is larger and has more dimensions compared to the nasa data-set. Figure 4.5 below shows the number of computations for the colors data-set.

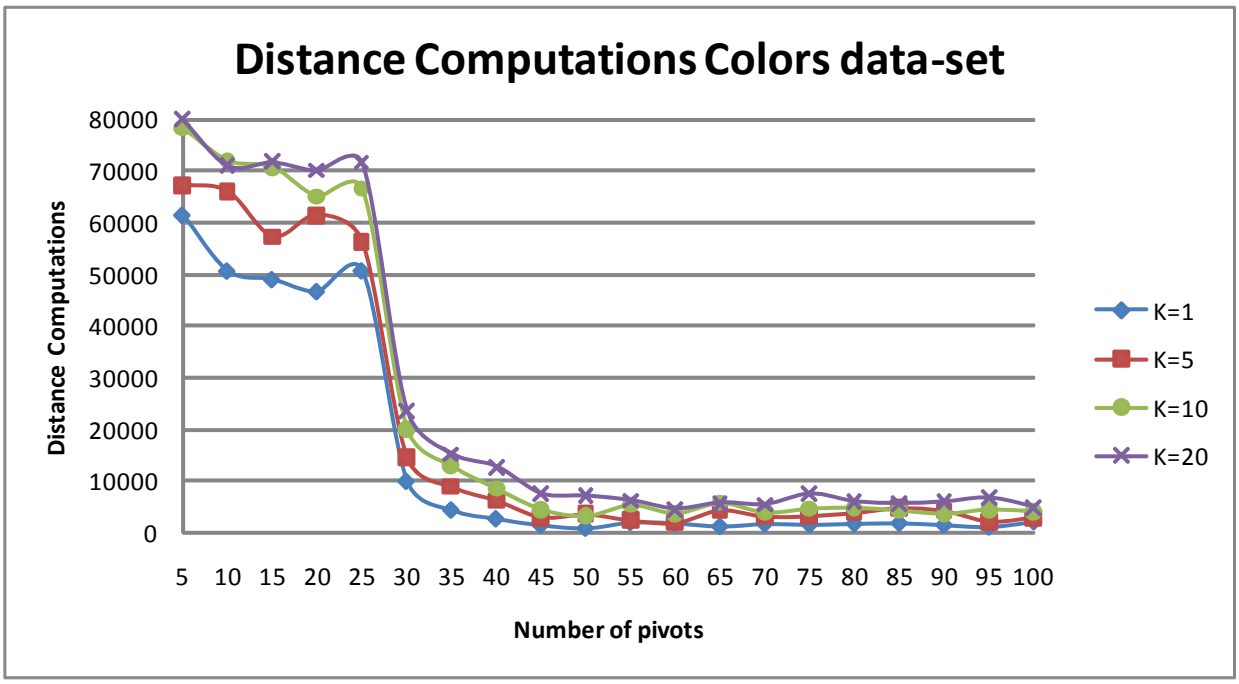


Figure 4.5: Number of distance computations for the colors data-set with growing number of pivots and with four different K values.

If we compare this result with the same experiment on the nasa data-set (figure 4.2), the difference is quite huge. The colors data-set clearly favors a pivot count higher than 30 but also stays very consistent all the way up to 100 pivots for all four K-values. Lets take a look how this fares with response times. Figure 4.6 contains the values for time spent building and sorting lower bounds for the colors data-set.

Number of pivots	5	10	15	20	25	30	35	40	45	50
Time building lowerbounds (ms)	772	1220	1478	1519	1960	2186	2864	3716	4221	5727

Number of pivots	55	60	65	70	75	80	85	90	95	100
Time building lowerbounds (ms)	5427	5694	6394	7165	11197	9375	9941	15301	10652	17065

Figure 4.6: Time spent on building and sorting lower bounds with varying numbers of pivots.

The cost for managing lower bounds has not changed much. There is a slight difference when using many pivots though.

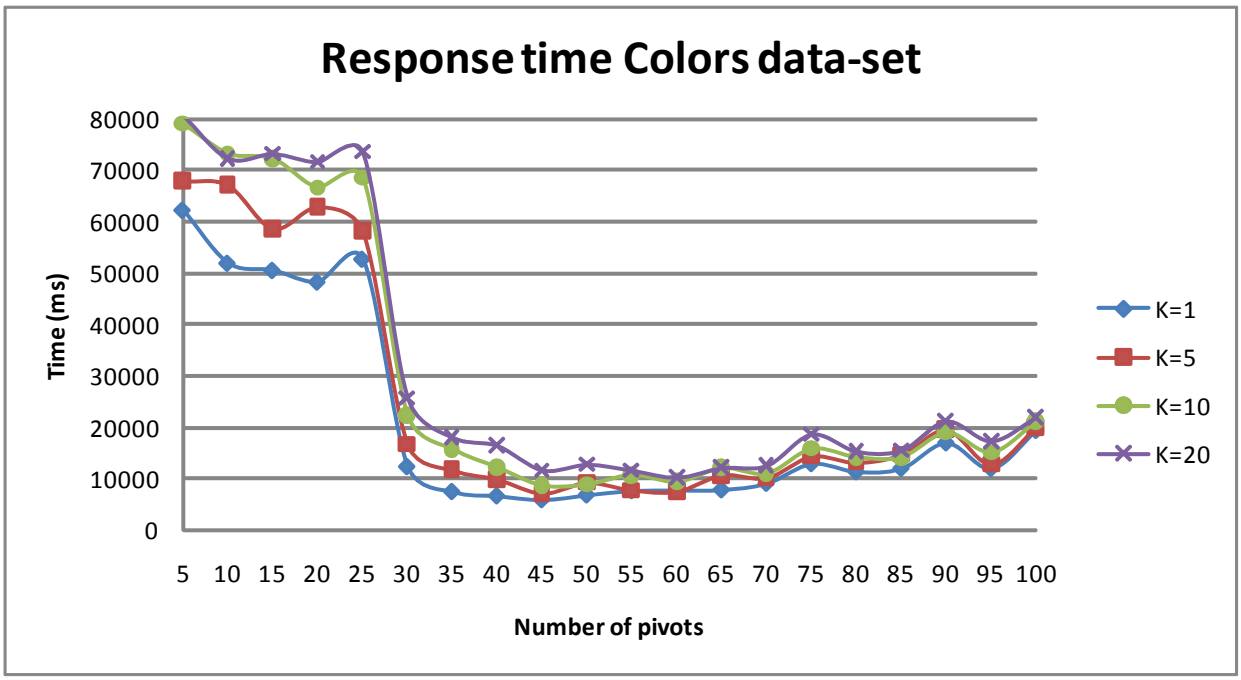


Figure 4.7: Response time for colors data-set with 1ms distance calculations

Now we are dealing with larger numbers of distance computations and time spent on lower bounds becomes more suppressed. This is why the curves looks very similar to distance computations chart(Figure 4.5).

Chapter 5

Conclusion and Future Research

5.0.1 Conclusion

This project has been a contribution to the larger NTNUStore project. The goal of this work has been to implement the original LAESA in NTNUStore and has focused on KNN search. There has been some difficulties with implementing the original pseudo code. Because of this and the time constraint, a decision to simplify the algorithm slightly was taken. As such, this project was not a 100% success. Despite this fact, the implemented algorithm is not very different from the original so labeling the project a partly success would be a fair assessment.

NTNUStore contained existing methods for solving KNN search via range search. The big question is if these methods have been improved. When comparing results from this projects with results presented in Erik Bagge Ottesen's thesis [10], it is difficult to nail a definite verdict. The reason for this is that the results are not very different and the fact that the results come from different environments with different system specifications. One advantage with the new algorithms presented is that one is no longer required to provide a range when performing a KNN search. For the end user it is more intuitive to not provide a range in a KNN search although, as mentioned earlier, this can be done automatically by the intrinsic dimensionality. In Erik Bagge Ottesen's thesis [10] it was shown that the number of distance computations in KNN search can quickly rise in numbers the further off the range is from the perfect range. How accurately the intrinsic dimensionality provide a range in the general case has not been studied in this project so it remains unclear if the benefit is significant or not.

5.0.2 Future Research

Some pieces of the implementation was made in a bit of a hurry but should not require extensive work to improve upon. Not just with respect to performance but also for better overview and integration in NTNUStore. This section will discuss

these issues in more detail and then widen the view to discuss other research that is not directly tied to this project.

As mentioned earlier the pivot information is stored in separate random access files. This information should be provided by the indexes themselves because this solution will require less code(the random access files will be obsolete). Furthermore it gives better overview and perhaps less disk-accesses since we can just retrieve the pivot information at the same time as we scan the index.

To store the lower bounds in memory the java hashmap was used. The choice fell on this because of its built-in feature that connects key to data. At one point this hashmap had to be sorted based on its values and this is not really intended by the library(the library only provides sorting based on the key). This was solved with the class *MapValueSort.java*. It is unclear if this procedure is a potential time sink so it is definitely something that should be analyzed further.

The algorithms currently accepts only the Btree. It should not be difficult to expand this to multiple access structures. Additionally it lacks generalization with respect to which metric is used and is currently hard coded with the L2Distance and the QFDistance.

It might be possible to reduce the amount of classes in the NTNUStore framework for better overview. The B-tree access structure, for example, has two different classes: *BTree.java* and *LaesaBtree.java* where the last extends the first. The LaesaBtree uses the distance as key, that is $d((p, o_i), k_i)$ where k_i is the objects key. The Btree stores in opposite order(an actual key is used as key in the index). This might be a bit confusing for researchers that are not familiar with NTNUStore beforehand. The suggestion then is to merge these classes into a single class that provides the combined features and also provides a choice to store with the actual key as the key or with the data as the key. If this is not feasible, then LaesaBtree can be extended to provide this choice.

Lets take a step back now and look at the bigger picture. It would be interesting to see how methods based on other indexing techniques, i.e metric balls, would perform in NTNUStore. Methods that uses disk-based storage exists so it should be possible to implement. If one would then have more data-sets that simulates different real world applications, one could do more extensive testing with these technologies. These tests could then possibly give results such as *the metric ball based method is fastest on data-set A* or range queries on data-set B is performed best by the pivoting algorithm A. Eventually one might have enough information to feed an optimizer that will make the best choice for a given situation automatically.

Speaking of optimizers, developing an optimizer for NTNUStore could be a very interesting project. Choosing the right algorithm for a certain job is not the only thing one can optimize. The respective algorithms can be optimized themselves and this can on its own be a big enough task for a project. As we have seen there are several factors that decides how fast a query will complete: Dimensionality,

data-set distribution, the size of K(how many neighbours is searched for) and the database size.

Chapter 6

References

[1] Tolga Bozkaya and Meral Ozsoyoglu. Distance-based indexing for high-dimensional metric spaces. In *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, SIGMOD '97, pages 357–368, New York, NY, USA, 1997. ACM.

[2] Tolga Bozkaya and Meral Ozsoyoglu. Distance-based indexing for high-dimensional metric spaces. *SIGMOD Rec.*, 26:357–368, June 1997.

[3] Svein Erik Bratsberg and Magnus Lie Hetland. Scaling metric indexing and search by applying database techniques. 2000.

[4] S. Brin. Near neighbor search in large metric spaces. In *21th International Conference on Very Large Data Bases (VLDB 1995)*, 1995.

[5] Paolo Ciaccia Deis, Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree: An efficient access method for similarity search in metric spaces. pages 426–435, 1997.

[6] Magnus Lie Hetland. The basic principles of metric indexing. *Swarm Intelligence for Multi-objective Problems in Data Mining*, 2009.

[7] H. V. Jagadish, Beng Chin Ooi, Kian-Lee Tan, Cui Yu, and Rui Zhang. idistance: An adaptive b+-tree based indexing method for nearest neighbor search. *ACM Trans. Database Syst.*, 30:364–397, June 2005.

[8] 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*, pages 9–17, 1994.

[9] Francisco Moreno-Seco, Luisa Micó, and José Oncina. Extending laesa fast nearest neighbour algorithm to find the k nearest neighbours. In *Proceedings*

of the Joint IAPR International Workshop on Structural, Syntactic, and Statistical Pattern Recognition, pages 718–724, London, UK, UK, 2002. Springer-Verlag.

[10] Erik Bagge Ottesen. Similarity search in large databases using metric indexing and standard database access methods. *MSc thesis at Department of Computer and Information Science, NTNU*, 2009.

[11] Enrique Vidal Ruiz. An algorithm for finding nearest neighbours in (approximately) constant average time. *Pattern Recognition Letters*, 4(3):145 – 157, 1986.

[12] Hanan Samet. *Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[13] Caetano Traina, Jr., Roberto F. Filho, Agma J. Traina, Marcos R. Vieira, and Christos Faloutsos. The omni-family of all-purpose access methods: a simple and effective way to make similarity search more efficient. *The VLDB Journal*, 16:483–505, October 2007.

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

[15] Jeffrey K. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Inf. Process. Lett.*, 40(4):175–179, 1991.

[16] Pavel Zezula, Giuseppe Amato, Vlastislav Dohnal, and Michal Batko. *Similarity Search - The Metric Space Approach*, volume 32. Springer, 2006.

[17] Donghui Zhang. NEUStore: A Simple Java Package for the Construction of Disk-based, Paginated, and Buffered Indices. 2005.

Appendix A

Original LAESA Source Code

```
public pivot laesaAlgorithm(Map O, Map P, Map D, List PK, FloatArrayData x) ↩
    throws IOException{
    int y = 0;
    int z = 0;
    O.remove(new IntKey(randomInt2));
    int nc;
    float d_star; //d*
        pivot p_star; //p*
        float dxs; //dxs
        L2Distance distanceType = new L2Distance();
        pivot q;
        pivot b;
        float gq;
        float gb;
        pivot s;
        float gp = 0.0F;
        Map G = new HashMap();

        System.out.println("pivot set" + P.keySet());

        //begin
        d_star = Float.POSITIVE_INFINITY;
        p_star = null;
        s = new pivot (new IntKey((Integer)PK.get(0)),(FloatArrayData)(P.get(new↩
             IntKey((Integer)PK.get(0)))));
        nc = 0;
        distanceType.initialize(x);
        int i = 0;

        while (O.size() > 0){
            int a = 0;
            int c = 0;
            int d = 0;
            int e = 0;
                i++;
                System.out.println("RUNDE " + i);
                dxs = distanceType.distance(s.getData());
                O.remove(s.getKey());
                nc++;
                //UPDATING p_star, d_star
                if(dxs < d_star){
                    p_star = s;
```

```
                d_star = dxs;
            }
            q = null;
            gq = Float.POSITIVE_INFINITY;
            b = null;
            gb = Float.POSITIVE_INFINITY;

            int i2 = 0;
            //ELIMINATING AND APPROXIMATING  LOOP
            Iterator it = O.entrySet().iterator();
            System.out.println("Total size " + O.size());
            int u = 0;
            while (it.hasNext()) {
                u++;
                Map.Entry pairs = (Map.Entry)it.next();
                i2++;
                //UPDATING G, IF POSSIBLE
                e++;

                if(P.containsKey(s.getKey())){
                    y++;
                    BTree distances = (BTree)D.get(s.getKey().value());
                    IntKey aa = (IntKey)pairs.getKey();
                    Float p_distance_s = (Float)distances.search((IntKey)↩
                        pairs.getKey()).value();
                    if(!G.containsKey(pairs.getKey())){G.put(pairs.getKey(),↩
                         0.0F);}
                    if((Float)G.get(pairs.getKey()) < Math.abs(p_distance_s ↩
                        - dxs) ){
                        G.put(pairs.getKey(), Math.abs(p_distance_s - dxs));
                    }
                }
                if((Float)G.get(pairs.getKey()) != null){
                    gp = (Float)G.get(pairs.getKey());
                }

                if(P.containsKey(pairs.getKey())){ //if pEB
                        //ELIMINATING FROM B
                    if (gp >= d_star ){//&& nc > PK.size()/20
                        it.remove();
                        c++;
                    }
                    else{
                        if(gp < gb){
                            //System.out.println("APPROXIMATING: SELECTING ↩
                                FROM B");
                            gb = gp;
                            b = new pivot((IntKey)pairs.getKey(), (↩
                                FloatArrayData)pairs.getValue());
                        }
                    }
                }
                else{
                    //ELIMINATING FROM P-B
                    if(gp >= d_star){
                        //System.out.println("ELIMINATING FROM P-B");
                        it.remove();
                        c++;
                    }
                    //APPROXIMATING: SELECTING FROM P-B
                    else{
                        if(gp < gq){
                            //System.out.println("APPROXIMATING: SELECTING ↩
                                FROM P-B");
                            gq = gp;
                            q = new pivot((IntKey)pairs.getKey(), (↩
                                FloatArrayData)pairs.getValue());
```

```
                            }
                            else{z++;}
                        }
                    }
                }
                System.out.println("Antall ganger i while " + u);
                System.out.println("Number of records deleted " + c);
                System.out.println("Number of records left " + O.size());
                if(b != null){
                    s = b;
                }
                else{s = q;}
        }
        System.out.println("Antall ganger inn i while " + y);
        System.out.println("z =  " + z);
        p_star.setDistance(d_star);
        return p_star;
    }
```

Appendix B

Final LAESA Source Code

```
public class LaesaKNN {
    private static final String DATA_FILE = "data/colors";
    //private static final String DATA_FILE = "data/nasa";
    int randomInt;
    private static final int PAGE_SIZE = 8192;
    private static final int BUFFER_SIZE = 10000;
    private List<FloatArrayData> queryStrings;
    private List<FloatArrayData> queryPivots;
    private LRUBuffer buffer;
    private BTree<IntKey, FloatArrayData> dataFile;
    private BTree<IntKey, FloatArrayData> bp;
    private Map pivots;
    private Map allObjectsKeys;
    private List pivotKeys;

    //Number of neighbours
    private static final int k = 1;
    //private static final int k = 5;
    //private static final int k = 10;
    //private static final int k = 20;

    private static pivot[] result = new pivot[k];
    int distanceCalcs = 0;
    FloatArrayData nasa_sample = null;
    FloatArrayData colors_sample = null;
    long timerBuildLowerBounds;
    long timerBuildLowerBounds2;
    long timerBuildLowerBoundsTotal = 0;
    long timersortLowerBounds;
    long timersortLowerBounds2;
    //INDEX_TYPE: -1=HeapFile, 0=BTree
    private int INDEX_TYPE = -1;

    public void laesaPivotSelection(int bpCount,LRUBuffer buffer) throws ↩
        IOException, InterruptedException{

        System.out.println("************************************************");
        System.out.println("*                                              *");
        System.out.println("*            Pivot Selection                   *");
        System.out.println("*                                              *");
        System.out.println("************************************************");
        System.out.println("");
        System.out.println("You have chosen to use " + bpCount + " Pivots.");
```

```
        buffer.flush(null);
        buffer.clearIOs();
        pivots = new HashMap();
        pivotKeys = new ArrayList();
        Map distanceAccumulator = new HashMap();
        float distance = 0.0F;
        int i = 1;
        pivot b_marked = null;
        pivot b = null;

        Cursor<IntKey, FloatArrayData> c2 = dataFile.query(new IntKey(0), new ↩
            IntKey(Integer.MAX_VALUE));
        c2.next();
        b_marked = new pivot((IntKey)c2.getKey(),(FloatArrayData)c2.getData());
        c2 = null;
        int counter = 0;

        while(pivots.size() <= bpCount){
            counter++;

            LaesaIndex index = null;
            //HeapFile index = null;
            //LaesaBTree indexB;

            //System.out.println("ROUND: " + counter);
            Cursor<IntKey, FloatArrayData> c = dataFile.query(new IntKey(0), new↩
                 IntKey(Integer.MAX_VALUE));
            float maxDistance = 0;
            b = b_marked;
            Distance<FloatArrayData> initializeDistance = new L2Distance((↩
                FloatArrayData)b.getData());
            if(INDEX_TYPE == 0){
                index = new LaesaBTree<FloatArrayData>(buffer, "data/pivot" + b.↩
                    getKey() + ".BTree", initializeDistance);
            }
            else{
                index = new LaesaHeapFile(buffer, "data/pivot" + b.getKey() + ".↩
                    HeapFile", initializeDistance);
                //index = new HeapFile(buffer, "data/pivot" + b.getKey() + ".↩
                    HeapFile", true, new IntKey(0), colors_sample);
            }
            while (c.next()){
                while(pivots.containsKey(c.getKey())){c.next();}
                distance = index.insert2((IntKey)c.getKey(), (FloatArrayData)c.↩
                    getData());
                if(distanceAccumulator.containsKey(c.getKey())){
                    float temp = (Float)(distanceAccumulator.get(c.getKey()));
                    distanceAccumulator.put(c.getKey(), temp+distance);
                }
                else{distanceAccumulator.put(c.getKey(), distance);}
                if((Float)(distanceAccumulator.get(c.getKey()))> maxDistance){
                    b_marked.setKey((IntKey)c.getKey());
                    b_marked.setData((FloatArrayData)c.getData());
                    maxDistance = (Float)distanceAccumulator.get(c.getKey());
                }
            }

            pivots.put(b_marked.getKey(), b_marked.getData());
            pivotKeys.add(b_marked.getKey().value());


            index.close();
            buffer.flush(null);
            i++;
        }
        /*
```

```
        HeapFile inde = new HeapFile(buffer, "data/pivotIntKey[value=27978].↩
            HeapFile", false, new IntKey(0), new FloatData(0.0F));
        Cursor cc = inde.fullScan();
        while(cc.next()){
            System.out.println(cc.getKey() + " data " + cc.getData());
        }*/

        if(!pivotKeys.contains(1)){new File("data/pivotIntKey[value=1]").delete↩
            ();}
        pivots.remove(b_marked.getKey());
        pivotKeys.remove(pivotKeys.size()-1);
        System.out.println("pivots selected: " + pivots.keySet());
        ObjectOutputStream output;

        try // open file
        {
            output = new ObjectOutputStream(new FileOutputStream( "data/pivot.↩
                keys" ) );
            output.writeObject(pivotKeys);
            output.close();
            output = new ObjectOutputStream(new FileOutputStream( "data/pivots.↩
                hashmap" ) );
            output.writeObject(pivots);
            output.close();
        } // end try
        catch ( IOException ioException )
        {
            System.err.println( "Error opening file." );
        } // end catch

    }


    private void readData() throws IOException {

        List<FloatArrayData> fileData = TestUtil.readFloatDataFromFile(DATA_FILE↩
             + ".data");
        buffer = new LRUBuffer(BUFFER_SIZE, PAGE_SIZE);
        dataFile = new BTree(buffer, "data/test", true, new IntKey(0), fileData.↩
            get(0));
        int key = 0;
        for (FloatArrayData data : fileData) {
            key++;
            dataFile.insert(new IntKey(key), data);
        }
        dataFile.close();
    }

    static Map sortByValue(Map map) {
         List list = new LinkedList(map.entrySet());
         Collections.sort(list, new Comparator() {
              public int compare(Object o1, Object o2) {
                   return ((Comparable) ((Map.Entry) (o1)).getValue())
                  .compareTo(((Map.Entry) (o2)).getValue());
              }
         });

        Map result = new LinkedHashMap();
        for (Iterator it = list.iterator(); it.hasNext();) {
            Map.Entry entry = (Map.Entry)it.next();
            result.put(entry.getKey(), entry.getValue());
        }
        return result;
    }
  //O = all Objects, P = pivots, D = pivot-object-distances, PK=pivot-keys, x = ↩
      test-object k = number of nearest neighbors
```

```
@SuppressWarnings({ "rawtypes", "unchecked" })
public pivot[] laesaAlgorithm(Map O, Map P, Map D, List PK, FloatArrayData x↩
    , float[][] matrix, int k) throws IOException{
    System.out.println("************************************************");
    System.out.println("*                                              *");
    System.out.println("*            Laesa KNN Algorithm               *");
    System.out.println("*                                              *");
    System.out.println("************************************************");
    System.out.println("");

    O.remove(new IntKey(randomInt));
    float dxs; //dxs
    //QFDistance distanceType = new QFDistance();
    L2Distance distanceType = new L2Distance();
    distanceType.initialize(x);
    Map G = new HashMap();
    Map dxsAll = new HashMap();
    float pmin = Float.POSITIVE_INFINITY;;
    float gmin = 0.0F;
    IntKey Resultkey = null;
    FloatArrayData ResultData = null;
    float bestRealDistanceSoFar = Float.POSITIVE_INFINITY;
    pivot init = new pivot(null,null);
    init.setDistance(Float.POSITIVE_INFINITY);
    result[0] = init;
    int added = 0;
    int counter2 = 0;
    pivot candidate = null;

    //if no pivots used
    if(PK.size() == 0){
        Iterator it = O.entrySet().iterator();
        while (it.hasNext()) {
            counter2++;
            Map.Entry pair = (Map.Entry)it.next();
            float dpx = distanceType.distance((FloatArrayData)O.get(pair.↩
                getKey()));
            distanceCalcs++;
            //if NN search
            if(k == 1){
                if(dpx < pmin){
                    ResultData = (FloatArrayData)O.get(pair.getKey());
                    Resultkey = (IntKey)pair.getKey();
                    candidate = new pivot(Resultkey, ResultData);
                    pmin = dpx;
                }
            }
            //if KNN search and result array is not filled up
            else if((added < k) || (result[k-1] != null && dpx < result[k↩
                -1].getDistance())){
                ResultData = (FloatArrayData)O.get(pair.getKey());
                Resultkey = (IntKey)pair.getKey();
                candidate = new pivot(Resultkey, ResultData);
                candidate.setDistance(dpx);
                if(added < k){
                    for (int a = 0; a < k ; a++){
                        if(result[a] != null && dpx < result[a].getDistance↩
                            ()){
                            for(int b = added+1; b > a && b <= k; b--){
                                    if(b == k ){b = 4;}
                                    result[b] = result[b-1];
                            }
                            result[a] = candidate;
                            break;
                        }
                    }
                }
```

```
                    //if KNN search and result array is filled up
                    else{
                        int counter = 0;
                        int k2 = k-1;
                        if(dpx <= result[k-1].getDistance()){
                            for(int b = k2; b > 0; b--){
                                if(dpx <= result[b-1].getDistance()){
                                    counter++;
                                }
                                else{break;}
                            }
                            for(int c = 0; c < counter; c++){
                                int indeks = (k-1)-c;
                                result[indeks] = result[indeks-1];
                            }
                            result[k2 - counter] = candidate;
                        }
                    }
                    added++;
                }
                if(k == 1){
                    candidate.setDistance(pmin);
                    result[0] = candidate;
                }
            }
        }
        //If pivots used
        else{
            timerBuildLowerBounds = System.currentTimeMillis();
            //go through pivots and compute distance to x(the test object)
            for(int ii = 1; ii <= P.size(); ii++){
                IntKey key = new IntKey((Integer)PK.get(ii-1));
                FloatArrayData data = (FloatArrayData)P.get(key);
                pivot BP = new pivot(key,data);
                dxs = distanceType.distance(BP.getData());
                distanceCalcs++;
                dxsAll.put(key, dxs);
            }
            //iterate through pivots
            Iterator itr = dxsAll.entrySet().iterator();
            long t1 = System.currentTimeMillis();
            BTree distancesB = null;
            HeapFile distancesH = null;
            Cursor testCursor = null;
            while(itr.hasNext()){
                Map.Entry pair = (Map.Entry)itr.next();
                IntKey BPkey = (IntKey)pair.getKey();
                Float dBPs = (Float)pair.getValue();
                int BPkeyi = BPkey.value();
                if(pmin > (Float)pair.getValue()){
                    pmin = (Float)pair.getValue();
                }
                if(INDEX_TYPE == 0){
                    distancesB = (BTree)D.get(BPkeyi);
                    testCursor = distancesB.query(new IntKey(1), new IntKey(↩
                        Integer.MAX_VALUE));
                }
                else{
                    distancesH = (HeapFile)D.get(BPkeyi);
                    testCursor = (HeapFileCursor) distancesH.fullScan();
                }
                Iterator it = 0.entrySet().iterator();
                int counter = 0;
                int counter1 = 0;
                long timer1_2 = 0;
                long timer3_4 = 0;
```

```
                //iterate through all objects and create lower bounds

                while (testCursor.next()) {
                    //System.out.println("test cursor : " + testCursor.getKey())↩
                        ;
                    //if the cursor contains the test object, skip to next
                    if(((IntKey)((testCursor.getKey()))).value() == randomInt){
                        testCursor.next();
                    }
                    long timer3 = System.currentTimeMillis();
                    while(dxsAll.containsKey(testCursor.getKey()) || testCursor.↩
                        getData() == null){
                        testCursor.next();
                    }
                    Float p_distance_s = ((FloatData)testCursor.getData()).value↩
                        ();
                    long timer2 = System.currentTimeMillis();
                    if(!G.containsKey(testCursor.getKey())){G.put(testCursor.↩
                        getKey(), 0.0F);}
                    if((Float)G.get(testCursor.getKey()) < Math.abs(p_distance_s↩
                         - (Float)pair.getValue())){
                        G.put(testCursor.getKey(), Math.abs(p_distance_s - (↩
                            Float)pair.getValue()));
                        if(Math.abs(p_distance_s - (Float)pair.getValue()) > ↩
                            gmin){
                            gmin = Math.abs(p_distance_s - (Float)pair.getValue↩
                                ());
                        }
                    }
                    //System.out.println("G: " + G);
                }
            }
            G.remove(new IntKey(randomInt));
            long t2 = System.currentTimeMillis();
            long timersortLowerBounds = System.currentTimeMillis();
            //sort the lowerbounds
            SortedMap sortedData = new TreeMap(new MapValueSort.ValueComparer(G)↩
                );
            sortedData.putAll(G);
            long timersortLowerBounds2 = System.currentTimeMillis();
            //MapValueSort sort = new MapValueSort();
            //sort.printMap(sortedData);
            Iterator it = sortedData.entrySet().iterator();
            timerBuildLowerBounds2 = System.currentTimeMillis();
            timerBuildLowerBoundsTotal = timerBuildLowerBoundsTotal + (↩
                timerBuildLowerBounds2 - timerBuildLowerBounds);
            long t3 = System.currentTimeMillis();


            //interate through lowerbounds and calculate result
            pmin = Float.POSITIVE_INFINITY;
            while (it.hasNext()) {
                counter2++;
                Map.Entry pairs = (Map.Entry)it.next();
                ResultData = (FloatArrayData)O.get(pairs.getKey());
                Resultkey = (IntKey)pairs.getKey();
                candidate = new pivot(Resultkey, ResultData);
                float dpx = 0.0F;

                //if NN search
                if(k == 1){
                    if((Float)sortedData.get(pairs.getKey()) > pmin){
                        System.out.println("lowerbound > best distance found so ↩
                            far, breaking...");
                        break;
                        }
```

```
                    if(dxsAll.containsKey(pairs.getKey())){pairs = (Map.Entry)it↩
                        .next();}
                    else{
                        //System.out.println("counter: " + counter2 + " pairs.↩
                            getKey() " + pairs.getKey() + " O.get " + O.get(↩
                            pairs.getKey()));
                        dpx = distanceType.distance((FloatArrayData)O.get(pairs.↩
                            getKey()));
                        distanceCalcs++;
                        if(dpx < pmin){
                            pmin = dpx;
                            candidate.setDistance(pmin);
                            result[0] = candidate;
                        }
                    }
                }
                //if KNN search and result array is not filled up
                else if((added < k)){
                    dpx = distanceType.distance((FloatArrayData)O.get(pairs.↩
                        getKey()));
                    distanceCalcs++;
                    candidate.setDistance(dpx);
                    for (int a = 0; a < k ; a++){
                        if(result[a] != null && dpx < result[a].getDistance()){
                            for(int b = added+1; b > a && b <= k; b--){
                                    if(added == 2 ){System.out.println("b " + b)↩
                                        ;}
                                    if(b == k ){b = k-1;}
                                    result[b] = result[b-1];
                            }
                            result[a] = candidate;
                            added++;
                            break;
                        }
                    }
                }
                //if KNN search and result array is filled
                else{
                    if((Float)sortedData.get(pairs.getKey()) > result[k-1].↩
                        getDistance()){
                        System.out.println("stopped after going through " + ↩
                            counter2 + " objects");
                        break;
                    }

                    dpx = distanceType.distance((FloatArrayData)O.get(pairs.↩
                        getKey()));
                    distanceCalcs++;
                    candidate.setDistance(dpx);
                    int counter = 0;
                    int k2 = k-1;

                    if(dpx <= result[k-1].getDistance()){
                        for(int b = k2; b > 0; b--){
                            if(dpx <= result[b-1].getDistance()){
                                counter++;
                            }
                            else{break;}
                        }
                        for(int c = 0; c < counter; c++){
                            int indeks = (k-1)-c;
                            result[indeks] = result[indeks-1];
                        }
                        result[k2 - counter] = candidate;
                    }
                }
            }
```

```
            long t4 = System.currentTimeMillis();
            System.out.println("Time spent calculating distances: " + (t4 - t3)↩
                /1000 + " Seconds.");

        }

        System.out.println("Total number of Objects " + O.size());
        System.out.println("Total distancecalculations: " + distanceCalcs);
        return result;
    }



    public static void main(String[] args) throws Exception {
        LaesaKNN test = new LaesaKNN();
        float[] n_sample = {-0.214579F, 0.224219F, -0.123123F, 0.0759739F, ↩
            0.433112F, 0.23946F, -0.0233754F, 0.327327F, 0.0959396F, 0.115914F, ↩
            0.0110738F, -0.16896F, -0.0705952F, 0.306239F, 0.263311F, -0.0360219↩
            F, -0.0268898F, 0.164756F, 0.121092F, -0.0319838F};
        test.nasa_sample = new FloatArrayData(n_sample);
        float[] c_sample = {0.1977F, 0.34111F, 0.0347584F, 0.01284F, 0.0F, 0.0F,↩
             3.6169E-5F, 0.0F, 0.0F, 0.0F, 0.0F, 0.0F, 0.0F, 0.0F, 0.00141059F, ↩
            7.2338E-5F, 0.0F, 0.0F, 0.0F, 0.0F, 0.0F, 0.0F, 0.0F, 0.0F, ↩
            0.0584852F, 0.0F, 0.0210142F, 0.0616319F, 0.0F, 0.0F, 0.0F, 0.0F, ↩
            0.0F, 0.0F, 0.0F, 0.0F, 0.0F, 0.0F, 0.00669126F, 0.0010489F, 0.0F, ↩
            0.0F, 0.0396412F, 0.0F, 0.0279225F, 0.0240524F, 0.0F, 0.0F, 0.0F, ↩
            0.0F, 0.0F, 0.0F, 0.0F, 0.0F, 0.0F, 0.0F, 0.0F, 0.0F, 0.0F, ↩
            0.0F, 0.0F, 0.053928F, 0.00115741F, 0.0F, 0.0F, 0.0F, 0.0F, 0.0F, ↩
            0.0F, 0.0F, 0.0F, 0.0298756F, 0.0F, 0.00329138F, 0.0F, 0.0F, 0.0F, ↩
            0.0F, 0.0F, 0.0581236F, 0.00842737F, 0.0F, 0.0F, 0.0F, 0.0F, 0.0F, ↩
            0.0F, 0.0F, 0.0F, 0.0F, 0.0F, 5.78704E-4F, 0.0F, 0.0F, 0.0F, 0.0F, ↩
            0.0F, 0.0160229F, 1.80845E-4F, 0.0F, 0.0F, 0.0F, 0.0F, 0.0F, 0.0F, ↩
            0.0F, 0.0F, 0.0F, 0.0F, 0.0F, 0.0F};
        test.colors_sample = new FloatArrayData(c_sample);
        BTree allObjects;

        //array with number of pivots to be used
        int[] pivots = new int[20];
        pivots[0] = 5;
        pivots[1] = 10;
        pivots[2] = 15;
        pivots[3] = 20;
        pivots[4] = 25;
        pivots[5] = 30;
        pivots[6] = 35;
        pivots[7] = 40;
        pivots[8] = 45;
        pivots[9] = 50;
        pivots[10] = 55;
        pivots[11] = 60;

        pivots[12] = 65;
        pivots[13] = 70;
        pivots[14] = 75;
        pivots[15] = 80;
        pivots[16] = 85;
        pivots[17] = 90;
        pivots[18] = 95;
        pivots[19] = 100;

        for(int iii = 0; iii < pivots.length; iii++){

            BufferedWriter out2 = new BufferedWriter(new FileWriter("data/↩
                average" +  pivots[iii] + ".txt"));
            BufferedWriter out = new BufferedWriter(new FileWriter("data/↩
                lowerbounds" +  pivots[iii] + ".txt"));
```

```
            LRUBuffer buffer = new LRUBuffer(10000, 8192);
            int average = 0;
            int testRounds = 100;

            Random randomGenerator = new Random();
            if(DATA_FILE == "data/colors"){
                allObjects = new BTree(buffer, "data/colors_data_and_keys", ↩
                    false, new IntKey(0),test.colors_sample);
                test.dataFile = new BTree(buffer, "data/colors_data_and_keys", ↩
                    false, new IntKey(0), test.colors_sample);

            }
            else{
                allObjects = new BTree(buffer, "data/nasa_data_and_keys", false,↩
                    new IntKey(0),test.nasa_sample);
                test.dataFile = new BTree(buffer, "data/nasa_data_and_keys", ↩
                    false, new IntKey(0), test.nasa_sample);
            }

            TestUtil.deleteFiles();

            long timerCreatePivots = System.currentTimeMillis();
            test.laesaPivotSelection(pivots[iii], buffer);
            long timerCreatePivots2 = System.currentTimeMillis();

            for(int iiii = 1; iiii <= testRounds; iiii++){
                test.distanceCalcs = 0;
                System.out.println("Testround " + iiii + " of " + testRounds);
                long timerTotal = System.currentTimeMillis();

                if(DATA_FILE == "data/colors"){test.randomInt = randomGenerator.↩
                    nextInt(112682);}
                else{test.randomInt = randomGenerator.nextInt(40149);}
                /*
                int sz = 0;
                for (FloatArrayData query : fileData) {
                    sz = query.size();
                    break;
                }

                float[][] matrix = new float[sz][sz];
                // create identity matrix
                for (int i = 0; i < sz; i++) {
                    for (int j = 0; j < sz; j++) {
                        if (i == j)
                            matrix[i][j] = 1.0F;
                        else
                            matrix[i][j] = 0;
                    }
                }
                */

                FloatArrayData random = (FloatArrayData)(allObjects.search(new ↩
                    IntKey(test.randomInt)));
                System.out.println("Random object selected: " + random);
                System.out.println("With key: " + test.randomInt);
                Map pivotsHashmap = new HashMap();
                Map allObjectsKeys = new HashMap();
                List pivotKeys = new ArrayList();
                Map allObjectsMap = new HashMap();

                ObjectInputStream inputStream = null;
                ObjectInputStream inputStream2 = null;
                ObjectInputStream inputStream3 = null;


                try {
```

```
                    //Construct the ObjectInputStream object
                    inputStream = new ObjectInputStream(new FileInputStream("↩
                        data/pivots.hashmap"));
                    inputStream2 = new ObjectInputStream(new FileInputStream("↩
                        data/pivot.keys"));

                    Object obj = null;
                    Object obj2 = null;

                    while ((obj2 = inputStream2.readObject()) != null) {
                        if (obj2 instanceof ArrayList) {
                            pivotKeys = (ArrayList)obj2;
                        }
                        while ((obj = inputStream.readObject()) != null) {
                            if (obj instanceof HashMap) {
                                pivotsHashmap = (HashMap)obj;
                            }
                        }
                    }
                }
                catch (EOFException ex) { //This exception will be caught when ↩
                    EOF is reached
                    //System.out.println("End of file reached.");
                } catch (ClassNotFoundException ex) {
                    ex.printStackTrace();
                } catch (FileNotFoundException ex) {
                    ex.printStackTrace();
                } catch (IOException ex) {
                    ex.printStackTrace();
                } finally {
                    //Close the ObjectInputStream
                    try {
                        if (inputStream != null) {
                            inputStream.close();
                        }
                        if (inputStream2 != null) {
                            inputStream2.close();
                        }
                    } catch (IOException ex) {
                        ex.printStackTrace();
                    }
                }

                Map listOfBtreesHeapfiles = new HashMap();

                long timerOther3 = System.currentTimeMillis();
                for (int i = 0; i < pivotKeys.size(); i++){
                    //System.out.println("data/pivotIntKey[value=" + pivotKeys.↩
                        get(i));
                    if(test.INDEX_TYPE == 0){
                        BTree index = new BTree(buffer, "data/pivotIntKey[value↩
                            =" + pivotKeys.get(i) + "].BTree", false,new IntKey↩
                            (0),new FloatData(0.0F));
                        listOfBtreesHeapfiles.put(pivotKeys.get(i), index);
                    }
                    else{
                        HeapFile index = new HeapFile(buffer, "data/pivotIntKey[↩
                            value=" + pivotKeys.get(i) + "].HeapFile", false, ↩
                            new IntKey(0), new FloatData(0.0F));
                        Cursor c = index.fullScan();
                        /*
                        while(c.next()){
                            System.out.println(c.getKey() + " data " + c.getData↩
                                ());
                        }*/
                        listOfBtreesHeapfiles.put(pivotKeys.get(i), index);
                    }
```

```
                }

                Cursor allObjectsCursor = allObjects.query(new IntKey(1), new ↩
                    IntKey(Integer.MAX_VALUE));
                while (allObjectsCursor.next()){
                    allObjectsMap.put(allObjectsCursor.getKey(), ↩
                        allObjectsCursor.getData());
                }

                pivotsHashmap.remove(new IntKey(1));
                long timerOther4 = System.currentTimeMillis();
                long t1 = System.currentTimeMillis();
                System.out.println("");

                pivot[] p = test.laesaAlgorithm(allObjectsMap, pivotsHashmap, ↩
                    listOfBtreesHeapfiles,pivotKeys,random, null, k);

                long t2 = System.currentTimeMillis();
                for (int i = 0; i < result.length;i++){
                    System.out.println("Result[" + i + "] distance = " + result[↩
                        i].getDistance() + " key = " + result[i].getKey());
                }

                average = average + test.distanceCalcs;
        /*
                for(int i = 0; i < pivotKeys.size(); i++){
                    BTree tempindex = (BTree) listOfBtreesHeapfiles.get(↩
                        pivotKeys.get(i));
                    tempindex.close();
                }*/

                long timerTotal2 = System.currentTimeMillis();
                System.out.println("Time spent total : " + (timerTotal2 - ↩
                    timerTotal)/1000 + " Seconds.");
                System.out.println("Time spent creating pivot : " + (↩
                    timerCreatePivots2 - timerCreatePivots)/1000 + " Seconds.");
                System.out.println("Time spent building lowerbounds in total: " ↩
                    + test.timerBuildLowerBoundsTotal + " Milliseconds.");
                System.out.println("Time spent sorting : " + (test.↩
                    timersortLowerBounds2 - test.timersortLowerBounds)/1000 + " ↩
                    Seconds.");

            }
            int totalaverage = average / testRounds;
            System.out.println("Totalaverage: " + totalaverage + " average: " + ↩
                average);
            String averagestring = Integer.toString(totalaverage);
            int temp = iii+1;
            allObjects.close();
            out2.write("pivot number: " + pivots[iii] + " average: " + ↩
                averagestring);
            out2.newLine();
            out.write("data/lowerbounds" +  pivots[iii] + "  " + test.↩
                timerBuildLowerBoundsTotal + ".txt");
            out.newLine();
            out2.close();
            out.close();
        }

    }
}
```