Extended Cluster Pruning Algorithm for High-dimensional Retrieval

Frederik Martini[1] **and Nikolaj Mertz**[2]

[1]**frem@itu.dk**
[2]**nime@itu.dk**

May 2020

Course Code: BIBAPRO1PE

ABSTRACT

High-dimensional clustering is used by some content-based image retrieval systems to partition the data into groups; the groups (clusters) are then indexed to accelerate the processing of queries. In recent work, researchers at the IT-University of Copenhagen (ITU) have proposed and evaluated the extended cluster pruning algorithm (eCP) for high-dimensional retrieval. In this report, we propose a new C++ eCP implementation, created with an eye towards OS portability and code readability with the purpose of it being used for teaching. The algorithm has been evaluated using the ANN-benchmarks tool produced at ITU (`ann-benchmarks.com`), focusing on the performance of the algorithm when run against multiple sets of various sizes. The implementation performed well on the smaller data sets included in ANN-Benchmarks, competing with some state-of-the-art algorithms. The public eCP repository can be found at `github.com/nimertz/eCP`

Keywords: Approximate Nearest Neighbor, Clustering, eCP, ANN-Benchmarks

ACKNOWLEDGEMENTS

We would like to thanks our supervisor Associate Professor Björn Þór Jónsson for great feedback and discussions had during the duration of the project. We would also like to thank Martin Aumüller for meeting with us in person and showing us how to get started with using ANN-Benchmarks.

CONTENTS

1 INTRODUCTION

Nearest neighbor search is one of the most fundamental tools in many areas of computer science, such as image recognition, machine learning, and computational linguistics [1]. Unfortunately, a phenomenon called the curse of dimensionality [2] tells us that to obtain true nearest neighbors one must use linear time in the size of the dataset or time/space exponential to the dimensionality of the dataset, preventing efficient and exact nearest neighbor search. Researchers have therefore in an attempt to obtain efficient algorithms, focused on allowing the returned neighbors to be an approximation of the true nearest neighbors. For evaluating these algorithms this has meant looking at how 'close', in some technical sense, the obtained approximate nearest neighbors (ANN) are to the true nearest neighbors.

Given the many different approaches to approximate nearest neighbor algorithms, it is only natural to wonder how they perform and compare in empirical settings. Many of the approaches to ANN algorithms already have solidly established implementations, meaning that a new algorithm can show its worth by comparing its performance to the many previous algorithms on a collection of standard benchmark data sets that pertain to certain quality measures. ANN-benchmarks[3] aims to make this possible by providing a constantly updated overview of the current state-of-the-art ANN algorithms. Currently, the tool includes several established tree-based, graph-based and hashing-based ANN algorithms, but relatively few cluster-based ones.

In recent work, Guðmundsson et al., proposed, evaluated, and applied the extended cluster pruning algorithm (eCP) for high-dimensional retrieval [4]. The algorithm has been implemented in various forms, both in C++ on multicore machines and using Hadoop[5, 6] and Spark[7] for cloud computing. eCP has only been evaluated for large-scale content-based image retrieval with hundreds of millions or even billions of images. Because of this, it would be relevant to study its in-memory performance on smaller established data sets with nearest neighbors ground truth data. Furthermore, several authors have presented their own eCP implementation, but they are usually not that easy to use and are therefore not considered suitable for teaching purposes.

Given the lack of cluster-based algorithms included in ANN-Benchmarks it is also natural to ask how eCP's performance would compare to the already included state-of-the-art tree, graph and hashing-based ANN algorithms.

In this project, we have taken the eCP algorithm description and rewritten it from scratch in C++ to provide an improved version in terms of OS portability and code readability to make it suitable for teaching purposes. During development we have evaluated the retrieval quality on established data sets of high-dimensional descriptors using ANN-Benchmarks to iteratively improve the algorithm. The final implementation was successfully embedded into ANN-Benchmarks and found to be able to compete with some of the state-of-the-art ANN algorithms.

Only gylfi might have done a public implementation

On the smaller randomly generated datasets

In this report, we first outline high-dimensional indexing, the extended cluster pruning algorithm and ANN-Benchmarks in chapter 2. Secondly, we describe the implementation of eCP and how it has been integrated into ANN-Benchmarks, focusing on improvements made during the development process in chapter 3. Immediately after we discuss the measured results and their interpretation in chapter 4, arriving at a conclusion and discussion regarding the future of eCP in the context of ANN-Benchmarks in chapter 5.

2 BACKGROUND

Multimedia database systems manage and manipulate content-rich data types such as video, image, and audio. Unlike more conventional database applications, multimedia applications often require retrieval of data that are similar in characteristics, such as color, shape, and texture content, to a given query object. Therefore, the general approach to content-based retrieval has been to transform the contents information into a form supported by an indexing method (feature extraction) [8]. The most common approach is to associate their characteristics to points in a multi-dimensional feature space. In practice this means each feature can be represented as a vector consisting of d values corresponding to coordinates in a d-dimensional space. In addition feature transformation, like normalization or median filling, is often used to transform these feature vectors with the purpose of improving the accuracy of algorithms.

An example of feature vectors could be the ones within the Fashion-MNIST[1] data set, modeling Zalando's article images of clothing. It is often used for benchmarking machine learning algorithms and

[1] https://github.com/zalandoresearch/fashion-mnist

contains 70.000 feature vectors with 784 dimensions. The set is split into a training set of 60.000 feature vectors and test set of 10.000 with the purpose of being able to evaluate whether an algorithms fits the data properly. Each feature vector in the data set represents a 28x28 grayscale image associated with a clothing type label from 10 classes. These grayscale images were first extracted from raw full-size image data into the 784 dimensional feature vectors suitable for describing their characteristics.

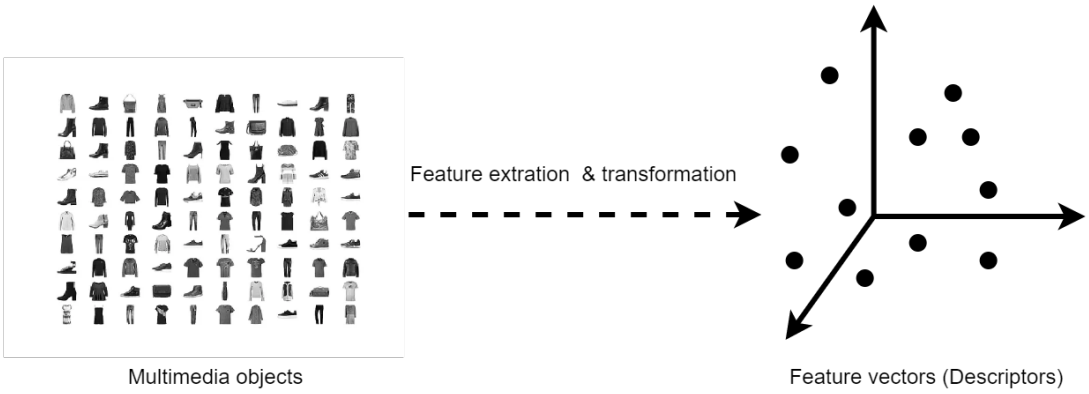

Figure 1. How multimedia is extracted and transformed to points in high-dimensional space

Quantifying similarity between multi-dimensional features involves the use of distance metrics. A simple yet common distance metric is the Euclidean distance, which as a function can be defined as:

$$dist\,(p,q) = \sqrt{\sum_{i=1}^{d} (q_i - p_i)^2} \tag{1}$$

where points are denoted as q_i, p_i and the dimensions of the space the points lie in d. The closer the Euclidean distance function's result is to 0 the more similar two points are. The Euclidean distance is considered the 'straight line distance' between two d-dimensional points in d-dimensional space. It essentially computes a scalar that determines how close the two points are to each other, and can be seen as d on figure 2. Another common distance metric is the angular distance, which like the Euclidean distance also computes how similar two points are. This however does so in a completely different way, not taking the magnitude of the vectors into account. The angular distance as the name implies, calculates the angle between two non-zero vectors as a scalar between 0 and 1, where 0 is identical and 1 is the negative vector. It can be seen as θ on figure 2 and is calculated by using the formula[2]

$$angular\ distance = \frac{cos^{-1}(cosine\ similarity)}{\pi} \tag{2}$$

where the cosine similarity between vector A and B is given by

$$cosine\ similarity = \frac{A \cdot B}{|A||B|} \tag{3}$$

There are different approaches to querying for content with similar characteristics in data sets of points in d-dimensional space depending on the application needs. There are range queries, where you wish to find all points whose values fall within certain given ranges defined as:

$$p \in S | p \in queryWindow \tag{4}$$

[2] https://en.wikipedia.org/wiki/Cosine_similarity

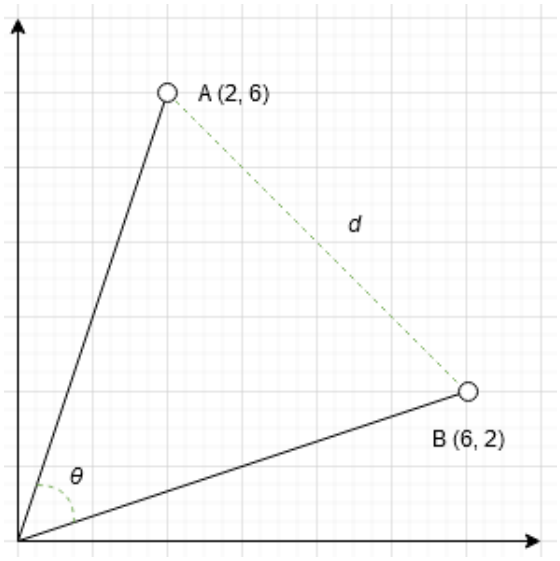

Figure 2. Euclidean and angular distance applied on two 2-dimensional feature vectors

where *queryWindow* is hyper rectangular window range query. If the *queryWindow* is defined as $[l_i, h_i]$ where $(0 <= i < d)$ and p values is represented as $x_i (0 \leq i < d)$, $p \in queryWindow$ is just $[l_i \leq x_i \leq h_i]$, which is often referred to as a window query. In addition there are similarity range queries where we are interested in finding all points which are within a given distance ε from a given point. This can be expressed as:

$$p \in S | dist(q,p) \leq \varepsilon \tag{5}$$

where q is the given query point and *dist* is the applied distance function. Lastly, there are K-nearest neighbor (KNN) queries which this report focuses on. Here we are interested in finding the k-most similar points in the data set which are closest in distance to a given query point. This is expressed as follows:

$$\{p_1 \cdots p_k \in S | \neg \exists p' \in S | \{p_1 \cdots p_k\} \wedge \neg \exists i, 0 < i \leq k : dist(p_i, q) > dist(p', q)\} \tag{6}$$

In case of ties in distance, more than k answers are allowed or points with similar distance are picked randomly to make up the k answers.

2.1 High-Dimensional Indexing

Given the large size of multimedia objects and the often large quantity, efficient and effective indexing methods are necessary to facilitate fast searching. To support fast similarity queries, a high-dimensional index is used to index the data points, and using this index as the underlying data structure, efficient retrieval algorithms are designed. A distance metric is often used to guide in building the index structure, while also allowing for efficiently searching on the index structure. The basic idea is to use the properties of similarity not only to build a tree-like structure but also to be able to prune "branches" in the processing of queries [8]. Indexing in a metric space naturally excludes the support for range queries given that points are not indexed based on their respective values.

The use of a distance metric on high-dimensional points suffers from the curse of dimensionality. It states that when the dimensionality increases, the volume of the space increases so rapidly that the available data becomes sparse. The contrast between points in terms of distance is very low in high-dimensional space and as the dimensionality increases, the difference between the nearest data point and the furthest data point reduces greatly. For KNN this means that the distance to the nearest neighbor approaches the distance to the furthest neighbor [2].

Local intrinsic dimensionality

If the nearest neighbors to a query point is consistently much closer to the query point than other points in the data set, then the data set is said to have high contrast. With low contrast, data sets suffer from vanishing variance and instability of nearest neighbors, which makes the construction of meaningful

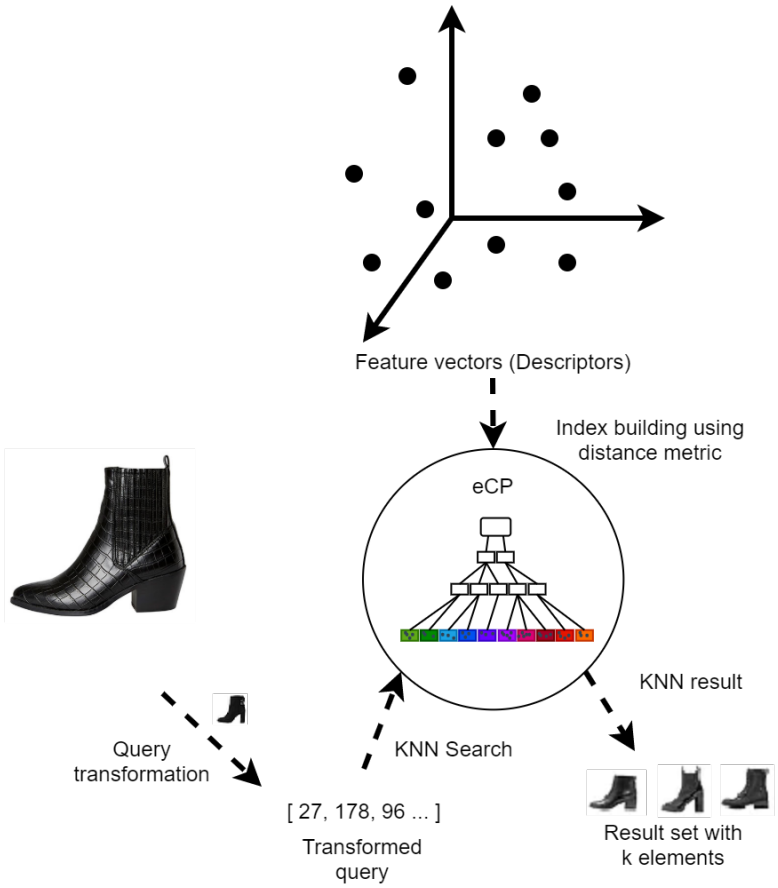

Figure 3. The general approach to high-dimensional indexing for KNN search

result sets difficult. Research has shown that high-dimensional data sets must have some contrast to be considered indexable and their result sets worthy to draw conclusions from [9].

Hence, in order to maintain an accurate representation of the space, the amount of data needed grows exponentially with the dimensionality. To help mitigate these problems, one should be careful in the choice of the number of dimensions used for representing multimedia objects and consider different dimensionality reduction methods [8]. It is important to note that the curse of dimensionality does not have a solution and researchers continue to address it by presenting new approaches to mitigating it.

As briefly touched upon, to obtain true nearest neighbors in a KNN search, one has to use linear time in the size of the data set or time/space exponential in the dimensionality of the data set. As an alternative, this has led to approximate nearest neighbor (ANN) algorithms in applications where errors can be tolerated. The premise of ANN is that relevant answers in the data set are likely to be closer to a query point than irrelevant data. Put differently, if the contrast of the data set is high, it is unlikely that the approximation would remove results. ANN algorithms are great for tuning time-quality tradeoffs and enabling users in getting answers quickly while waiting for more accurate answers. ANN is therefore a widely accepted compromise between accuracy and response time. There exist different extensively researched approaches to ANN algorithms. Tree-based algorithms such as kd-trees organize and partition data points based on specific conditions. Graph-based algorithms build a graph where each vertex is associated with a data point and adjacent to its true nearest neighbor. Algorithms like locality-sensitive hashing (LSH) uses hashing to project data points into lower-dimensional space. Finally, Cluster-based ANN algorithms partition data into groups referred to as clusters and then index these to accelerate the processing of queries [4].

high contrast = large difference between nearest and rest

Formally ANN algorithms build an index for a data set S of n points. In a preprocessing phase, this index is created to support the following type of ANN queries: for a query point q and an integer k,

return a result set $R = \{p_1, ..., p'_k\}$ of $k' \leq k$ distinct points from S that are "close" to the query q. ANN algorithms generate R by refining a set $C \subset S$ of candidate points close to q by choosing the k nearest points among those using distance metrics. The set of the k true nearest neighbors for q in S are then denoted as $R^* = \{p^*_1, ..., p^*_k\}$. Usually the result set is sorted according to their distance to the query point q. For ANN algorithms recall is often the main quality measure for evaluating the result set R. Intuitively, recall is the ratio of the number of points in the result set that are true nearest neighbors to the number k of true nearest neighbors [3]. However, this definition is not always used since it does not take into account when distances are not distinct. To account for indistinct distances the following definition is used, that takes the distance of the k-th true nearest neighbor as threshold distance:

$$recall(R, R^*) = \frac{|\{p \in R | dist(p,q) \leq dist(p^*_k, q)\}|}{k} \tag{7}$$

Another quality measure often compared against recall, is the actual speed of querying, often measured in queries per second. There is of course a trade-off between obtainable recall and the querying speed that algorithm designers have to take into account when creating and proposing new algorithms.

2.2 Cluster Pruning

Clustering itself is the task of grouping a set of objects such that objects in the same cluster are more similar to each other than those in other clusters. It is a common technique for static data analysis and used in many fields such as machine learning, data mining, bioinformatics, and image analysis [1]. Clustering can be achieved through various algorithms that differ in their definition of what constitutes a cluster and how to search for them. Therefore, clustering can be perceived as a multi-objective optimization problem of finding the appropriate clustering algorithm and parameter setting for the individual data set and intended use of results.

Parameter settings often include distance metrics, density threshold, or the number of expected clusters. An example of a clustering algorithm could be k-means clustering, which attempts to find the center of a natural cluster by recursively clustering a data set and finding a new centroid each time, repeating until convergence is reached. Inspired by k-means, Chierichetti et al. proposed an approach called 'Cluster Pruning' [10]. This method's simple preprocessing phase consists of selecting random cluster leaders from a subset of the entire data set and then clustering the rest of the points based on which leader they are nearest to. In the query processing phase, the nearest b clusters are searched and used to produce the approximate results. Based on this approach, Guðmundsson et al. studied cluster pruning in the context of large-scale image retrieval and proposed several parameter extensions for improving its performance [4].

Extended Cluster Pruning (eCP) like its predecessor has a preprocessing and query processing phase. During preprocessing random cluster representatives are selected, used for clustering the entire data set, and lastly leaders are chosen for each cluster.

The input to preprocessing are a set S of n high-dimensional descriptors $S = \{p_1, p_2, \ldots, p_n\}$ of dimension d. Normally, $l = \sqrt{n}$ random points $\{l_1, l_2, \ldots, l_n\}$ are selected from S as cluster leaders, and then all points in S are compared to them to create a subset for each. Each subset is a cluster and can be described as a tuple $[l_i, \{p_{i_1}, p_{i_2}, p_{i_v}\}]$ where l_i is the nearest leader and p_{i_1} to p_{i_v} are the points in the cluster

It was Chierichetti et al. that proposed using $\sqrt{n}$ leaders, since that number minimizes distance comparisons during the query phase, and would be optimal in terms of main-memory usage. Contrary to this, Guðmundsson et al. study on large-scale image indexing revealed that

$$l = \frac{n}{\lfloor desired\ cluster\ size / descriptor\ size \rfloor} \tag{8}$$

where desired cluster size is the IO granularity of the given operating system gave better results in a disk-based setting[4]. This value for l results in fewer but larger clusters, which naturally leads to a smaller index which is faster to search through.

After l clusters have been formed, a cluster representative needs to be selected, with the purpose of guiding the search to the cluster containing the nearest points. Chierichetti et al. presented three ways to select cluster representatives. The first was to simply have the leaders act as representatives. The second, to have representatives be the centroids of each cluster and third to have representatives be the

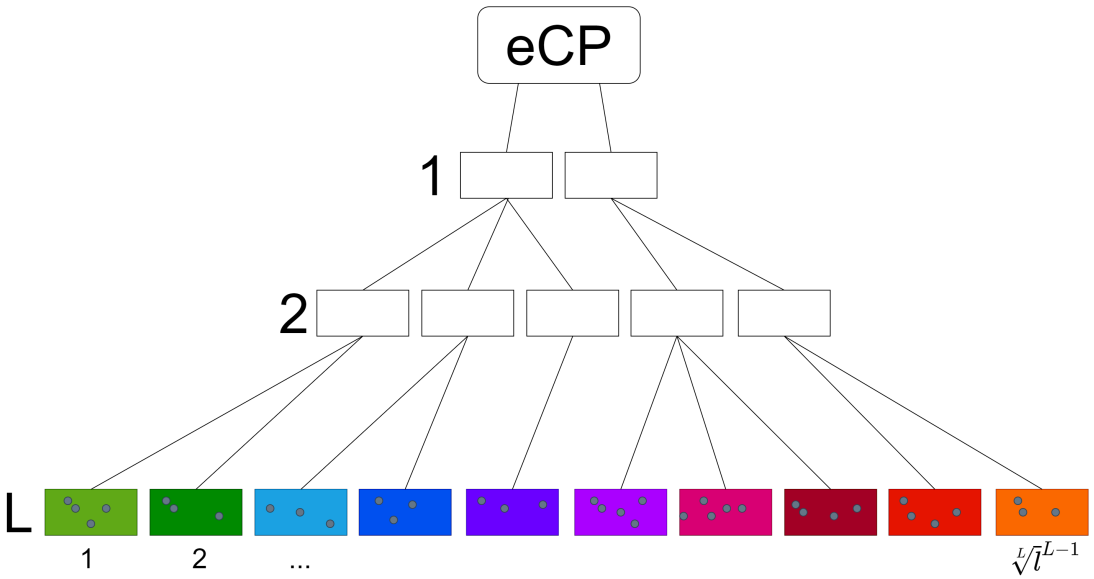

Figure 4. The eCP index structure

data point closest to the centroid of the cluster (medoid) [10]. Chierichetti et al. found that using centroid gave the best results, whereas Guðmundsson et al. found that even though centroids might give slightly better results, the difference in preprocessing time is too large for it to be worth it for large data sets. Guðmundsson et al. therefore argued that using leaders as representatives was a better choice, since the index structure would be finalized before descriptors are assigned to representatives. By knowing the cluster index beforehand, the set of descriptors can be clustered immediately resulting in a quicker preprocessing phase [4].

Depending on the L parameter, this cluster pruning method can be applied recursively to cluster the set of cluster leaders as just described. Given $L > 1$, the bottom level of the index then contains the l clusters gained from clustering S. From this bottom level $\sqrt[L]{l}^{L-1}$ descriptors are chosen to be leaders for the next level in the index. The new level will have $\sqrt[L]{l}$ fewer leaders than the bottom level. Next, descriptors from the bottom level are compared to the new leaders and $\sqrt[L]{l}^{L-1}$ clusters created. This process is repeated $L-1$ times until the top level is reached, resulting in a tree-like index structure as seen below, with fewer similarity comparisons needed for searching. Guðmundsson et al. found that for large high-dimensional data sets using $L = 2$ gave the best results[4].

Chierichetti et al. also presented the parameter a for preprocessing, giving the option of assigning each point to several cluster representatives. A value above 1 of course results in larger clusters since points are represented more than once. Guðmundsson et al. went on to describe how the a parameter wasn't viable for indexing large data sets and that using $a = 1$ gave the best results. This conclusion was partly due to the b parameter Chierichetti et al. also presented. This parameter defines how many nearest clusters should be searched during query processing. Guðmundsson et al. found that incrementing b gave better performance than incrementing a, due to smaller clusters fitting in memory [4]. Query processing for eCP is straightforward. When L and $b = 1$, the query point q is first compared to the set of l representatives to find the nearest one. Then, q is compared to all points in that representative's cluster to find its k nearest neighbors. For $L > 1$, the index is traversed from the top level, comparing distances until the bottom level is reached and the nearest cluster found.

2.3 ANN-Benchmarks

The ANN-benchmarking tool is an environment for comparing the current state of the art ANN algorithms in terms of both speed and accuracy. As the authors, E. Bernhardsson, M. Aumüller, and A. Faithfull describe themselves, there has not been many empirical attempts at comparing approaches to the ANN problem objectively and is why the tool was created. Researchers often evaluate their algorithm implementations based on a small set of competing algorithms and a small number of selected data sets [3].

Besides, researchers might find that competing ANN algorithm implementations use completely different conventions for input data and output of results. Certain papers might even omit details concerning certain required parameter settings. This makes it hard for readers to reproduce experimental results and the choice of data set and quality measures might appear biased towards the implementation. This is the problem the ANN-Benchmarking tool attempts to solve by providing a standard interface for measuring the performance achieved by ANN algorithms on different standard data sets and parameter settings. The only requirement of the implementation author is that the algorithm exposes a simple programmatic interface for building the data structure from training data and running queries on it. A run group specifying parameters for the algorithm must be provided as well [3].

The tool so far supports a wide variety of data sets ranging from a small set of 25 dimensions (Glove-25)[3] to a huge data set with 27983 dimensions (Kosarak[4]) and distance metrics such as Euclidean, angular, bit and hamming distance.

In terms of quality measures, ANN-Benchmarks can provide the index size of the data structure, index build time, the time it took to run a query and generate a set of k results, and the recall. ANN-Benchmarks uses the distance-based definition of recall seen in equation 7, that takes into account when distances are not distinct. ANN-Benchmarks specifically allows an easy way of creating interactive plots with "Queries per second"(y-axis, log-scaled) against "Recall" (x-axis). ANN-Benchmarks can also generate a website where graphs are interactive and data points show the algorithm parameters they were run with [3].

ANN-Benchmarks uses Docker[5] to create a container for each of the ANN algorithms, such that their dependencies do not interfere with one another and can run in isolation. This removes uncertainty by limiting the environment in which they run and makes it easier to add and remove algorithms for comparison.

3 IMPLEMENTATION

In this chapter we turn our attention to the C++ implementation of the algorithm. In section 3.1 the data structures are presented and in section 3.2 the implementation of the pre-processing phase accompanied by some of its pseudo-code is presented. In section 3.3 pseudocode describing how the nearest neighbors are obtained during the query-processing phase is outlined and in section 3.4, the implementation of the relevant distance functions are discussed. Section 3.5 describes how the eCP algorithm has been embedded into ANN-Benchmarks and section 3.6 discusses the performance improvements made throughout the development process. Section 3.7 talks about the coding style of the algorithm and finally section 3.8 sums up the most important points from this chapter

The implementation of the algorithm is based on the descriptions and pseudocode presented in Chierichetti et al. and Guðmundsson et al. papers [10, 4]. New approaches and pseudocode has been presented and discussed with the supervisor of the project, and as Chierichetti et al. outlines, the algorithm is split into a preprocessing and query-processing phase, something that the implementation structurally attempts to replicate. The a parameter that Chierichetti et al. proposed, dictating how many nodes each point is assigned was not implemented based on Guðmundsson et al. findings that incrementing the b parameter gave better performance than incrementing a, due to smaller nodes fitting in memory [4]. Looking through the data sets ANN-Benchmarks supports, it is clear that the two most common distance functions are the Euclidean, and angular distance and thus are the only ones supported in the implementation. It has been observed that the majority of the sets used in this project[6] use NumPy's float32 data type with the exception being the random-20 and random-100 data sets which use NumPy's float64 data type. Based on this, the C++ implementation has been adjusted to use floats instead of doubles to avoid unneeded precision.

3.1 Data Structure

The data structures used in this project have been implemented as C++ structs representing a single point, a node/cluster, and the entire index. The Point struct implementation contains its id, and its descriptor implemented as a float pointer. The `id` of the point is its index in the data set, which ANN-Benchmarks

[3] `https://github.com/stanfordnlp/GloVe`
[4] `http://fimi.uantwerpen.be/data/`
[5] `https://www.docker.com/`
[6] This was tested on glove-25-angular, mnist-784-euclidean, sift-128-euclidean

requires as output for calculating the recall. The `descriptors` member is implemented as a float pointer, pointing to the first attribute of the descriptor, so in essence it is an array with length kept globally.

Listing 1 Point Struct

```
struct Point {
        unsigned int id;
        float* descriptor;

        Point(float* _descriptor, unsigned int _id) {
                id = _id;
                descriptor = _descriptor;
        }
}
```

The Node struct has two lists storing its children and points respectively. The representative of the node is stored as the first element in the `points` member, and as mentioned in chapter 2, the representative is used for guiding the search by computing its distance to a query point. At the bottom level of the index, a node (cluster) will have an empty `children` member. The cluster can be viewed as a leaf of the tree-like index, containing points near to its representative contained within the same list.

Listing 2 Node Struct

```
struct Node {
        std::vector<Node*> children;
        std::vector<Point> points;

        Node(Point& p) {
                points.push_back(p);
        }
}
```

The index keeps track of the L parameter, the top-level from which the index is searched and lastly the data set in the form of points. Figuratively, the index can be viewed as a root pointing to the top level, which in turn is connected to the next level by its children and so on.

Listing 3 Index Struct

```
struct Index
{
        unsigned int L;
        std::vector<Node*> top_level;
        std::vector<Point> dataset;

        Index(unsigned int _L, std::vector<Node*>& _top_level,
         ↪  std::vector<Point>& _dataset)
        {
                L = _L;
                top_level = _top_level;
                dataset = _dataset;
        }
}
```

The data structures contain several pointers, in turn increasing the risk of memory leaks. This makes the inclusion of destructors essential. A destructor is created to clean up dynamically allocated memory like pointers. When a struct or class contains a pointer to dynamically allocated memory, a destructor should be written to release memory before the instance is destroyed. If this is not done it will result in a memory leak where the program will eventually run out of memory as the program piles up objects that consume memory but never releases it. The structs and their respective destructors can be found in `data_structures.hpp`

Vector Container

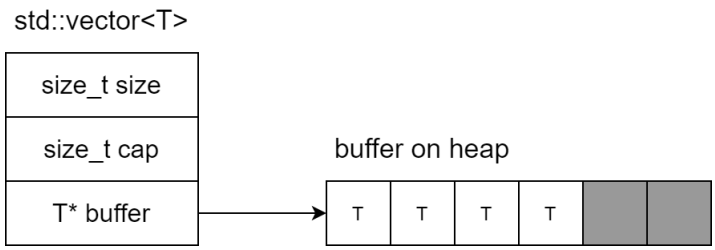

Figure 5. std::vector memory layout

The vector class from the standard C++ library has been used heavily in the implementation, since the algorithm revolves around scanning lists of unknown amounts of elements and comparing them. Because of this, it has been essential to figure out how to best utilize the container and when not to use it altogether. std::vector is a sequential container that can hold an arbitrary number of elements and is implemented as a dynamic array with a varying growth factor of around 2. This means that whenever the capacity is increased, the vector has to allocate memory for the new capacity and copy over the old elements. This happens when the size of the vector meets the capacity, as seen in figure 5. This of course comes at a cost, and it is therefore essential to only use a vector when the size of a list of elements is unknown at compile time. A key concept of the std::vector is that it allows for reserving space on the heap, to reduce reallocation and copying cycles. This is especially useful when building the eCP index, where a node on average represents $n^{1/(L+1)}$ points at the next level [4]. Furthermore, std::vector also exposes a `shrink_to_fit` function for releasing memory, hence aligning size and capacity. Any insert at the front of a vector is an $O(n)$ operation, since every element in the vector has to be copied to make room for the new entry [11]. This is important for the eCP implementation when inserting points into nodes and connecting the levels of the index.

3.2 Pre-Processing

The `eCP_Index` function located in `eCP.cpp` returns a pointer to the created index and is constructed following the pseudocode on algorithm 1. Index building is done by first calculating the size of the L levels in the index. Afterward the last partition, meaning the last level size, is used for creating the top level of nodes in the index. From there on, the $L-1$ levels of nodes are created and connected to the current bottom level by searching from the top level. When the nearest node at the bottom level has been found, the current node being built is added to its `children` list. This continues until the L level index has been built and all levels are connected. This simple approach to creating the index is only possible because the dataset is partitioned from the start, under the assumption that the order of descriptors is random. Furthermore, leaders are chosen as representatives so the entire index exists before assigning any points to clusters. Assigning points to clusters can therefore benefit from the index, dramatically reducing distance comparisons and therefore clustering time.

3.3 Query-Processing

Finding the b nearest clusters for a query point, has been implemented using the pseudocode in algorithm 2. For $L = 1$, it will return the b best clusters from the root/top level where the points are stored. For $L > 1$ it maintains a list of the b nearest nodes as it traverses the index $L-1$ times until the bottom level is reached. It makes use of the `scan_node` function to scan a node's children, meaning nodes at the next level, for the b nearest clusters to the query point, returning them in a list. The `scan_node` function itself makes use of a function during the scan to find the furthest node from the query point in the list of

Algorithm 1 CREATE-INDEX - `pre-processing.cpp`

```
1:  function CREATE-INDEX(dataset, L)
2:      // calculate level sizes bottom up
3:      L_level_sizes[L] ← ∅
4:      bottom_L ← dataset.size^(L/L+1)
5:      L_level_sizes[0] ← bottom_L
6:
7:      for i = 1 to L do
8:          level_size ← L_level_sizes[i − 1]^(L−i/L+1)
9:          L_level_sizes[i] ← level_size
10:     end for
11:
12:     // create top level
13:     top_level ← ∅
14:     for j = 0 to L_level_sizes[L-1] do
15:         node ← { dataset[j] }
16:         top_level.add(node)
17:     end for
18:     reverse level_size
19:     // build each level of nodes and insert them top down
20:     for level = 1 to L do
21:         for k to level_sizes[level] do
22:             // go down level - 1 levels from top_level
23:             nearest_bottom_level_node ← find_nearest_cluster(dataset[k], top_level, level - 1)
24:             node ← { dataset[k] }
25:             nearest_bottom_level_node.children.add(node)
26:         end for
27:     end for
28:
29:
30:     return top_level
31: end function
```

the nearest b nodes obtained so far. The furthest node is then swapped when a better candidate nearer the query point is found, and a new worst candidate is computed. This process continues for each level of the index until the b cluster at the bottom level has been found. These clusters will contain points that can be scanned for the k nearest neighbors (KNN).

Searching for k nearest neighbors at the bottom level of the index in the b nearest clusters found is straight forward. Each cluster is simply iterated over applying pseudocode similar to the one shown in algorithm 2. Here an accumulator in terms of a list is used once again to store the nearest points encountered so far. Each point is stored as a tuple of the id of the point and its distance to the query point. The distance is kept for calculating which nearest point met so far is the furthest away from the query point. This furthest point can then be swapped if a new point with a shorter distance is encountered. After a swap a new furthest point must be calculated from the list. Initially, the first k points are put into the list of nearest neighbor candidates, to ensure k or at least the amount of encountered points is obtained. When all points in the b clusters have been iterated over, the list of k nearest neighbors is returned.

Depending on the application, the KNN points can be sorted by the distance/similarity to the query point before being returned. This is required by ANN-benchmarks, so the C++ standard library sort, with a complexity of $O(N * log(N)$ comparisons[7] are applied to the result set of nearest neighbors. To avoid sorting two lists with regards to ids and distances, the nearest neighbor points are stored as tuples of ids and distances. Although recall can be evaluated on ids alone, distances are helpful for evaluation during development and data set analysis. In the source code, the list used for accumulating nearest points is a vector where capacity is reserved for k elements.

[7] `https://en.cppreference.com/w/cpp/algorithm/sort`

Algorithm 2 FIND-B-NEAREST-CLUSTERS - `query-processing.cpp`

```
1:  function FIND-B-NEAREST-CLUSTERS(top_level, query, b, L)
2:      // return the b best from top_level
3:      b_best ← ∅
4:      SCAN-NODE(query, node.children, b, b_best) // fills b_best
5:      while L > 1 do
6:          new_b_best ← ∅
7:          for node in b_best do
8:              SCAN-NODE(query, node.children, b, new_b_best)
9:          end for
10:         L ← L - 1
11:         b_best ← new_b_best
12:     end while
13:
14:     return b_best
15: end function
```

Algorithm 3 SCAN-NODE - `query-processing.cpp`

```
1:  function SCAN-NODE(query, nodes, b, b_best)
2:      furthest_node ← (-1, -1) //tuple of (index, distance)
3:      // find the furthest node if we have enough
4:      if | b_best | ≥ b then          not a scenario where we add more than b
5:          furthest_node = find_furthest_node(query, b_best)
6:      end if
7:
8:      for node in nodes do
9:          if | b_best | < b then
10:             b_best.add(node)
11:
12:             if | b_best | == b then
13:                 // we now have enough nodes, compute the worst to prepare to replace
14:                 furthest_node = find_furthest_node(query, b_best)
15:             end if
16:         else
17:             if distance(query, node.representative) < furthest_node.distance then
18:                 b_best[furthest_node.index] = node
19:                 furthest_node = find_furthest_node(query, b_best)
20:             end if
21:         end if
22:     end for
23: end function
```

3.4 Distance Functions

As the Euclidean and angular distance was found to be the most represented in ANN-Benchmarks, supporting these two functions would be important, allowing eCP to be compatible with more data sets and allow for a more complete algorithm evaluation. For making the introduction of new distance metrics to the algorithm easier, the distance function is kept as a global function pointer `g_distance_function`. This global variable is used during preprocessing and query-processing for all distance calculations between representatives and points.

The distance function implementations can be found in `distance.cpp` and is easily extended with new distance functions. The functions are required to adhere to the signature float* → float* → float, accepting two pointers each at the start of an array of floats, and returning the distance between the two points in some form. The distance returned must be positive and values approaching 0 must indicate more similar points. In an attempt to speed up distance comparisons, we have chosen to omit squaring sum in

F# like notation

the Euclidean distance (equation 1) resulting in equation 9 as we do not need the real distance.

$$euclidean\ distance = \sum_{i=1}^{d} (q_i - p_i)^2 \tag{9}$$

If the real distance is required, it is also possible to simply square the KNN search results before returning them. The same applies to the angular distance (equation 2) formula. Here we can omit dividing the result by π, as an approximation of the distance is sufficient for comparison as seen on equation 10

$$angular\ distance = cos^{-1}\left(\frac{A \cdot B}{|A||B|}\right) \tag{10}$$

3.5 ANN-Benchmarks Embedding

By design, ANN-Benchmarks requires a python module with a `fit` function that builds the index, and a `query` function that queries it. This is found in `eCP.py`, where the functionality adheres to the requirements outlined in the ANN-Benchmarks paper [3]. The implementation of `eCP_Index` in `eCP.cpp` receives the dataset, represented as a list of descriptors in the form of float lists, the index depth *L*, and the distance metric. It makes use of functions inside `pre-processing.cpp` to build the index bottom-up and insert the points top-down into it. This function also sets the global variables `g_vector_dimensions` indicating how many values a descriptor has, and the `g_distance_function` for descriptor comparisons during index creation and querying. Ensuring that the descriptor length is only stored once and not along with every descriptor is important for keeping the index data structure small. The `query` function receives the index it should run the query on, the query point represented as a float list, the *k* parameter and the *b* parameter

The ANN benchmarking tool has been the primary way of comparing and evaluating the eCP implementation against state-of-the-art ANN algorithms. As described in the 'Including your own algorithm' section of the ANN-Benchmarks repository[8], it is required to wrap the C++ project files into a python module. As recommended by ANN-Benchmarks, SWIG has been used to accomplish this. SWIG is a general wrapping tool allowing code from one language to be accessed in another language. This tool made it possible for C++ code to be used from within python by including the necessary function signatures and templates for mapping C++ templates to python, in the SWIG interface file.

ANN-Benchmarks uses a yaml file to configure the parameter settings supported by the embedded algorithms. This file dictates how the algorithms will run by including a docker file for retrieving the algorithm and making it available to the docker container. It must also include the module that receives the data set, the constructor of this python module, base-args containing the metric of the data set and finally run-groups specifying the parameter settings used on each run. ANN-Benchmarks outlines that algorithms must specify one or more "run groups", that each will be expanded into more lists of constructor arguments. In practice there are "args" which will be given to the constructor and "query-args" that are used to reconfigure the query parameters of the algorithm instance after its internal data structures have been built. The cartesian product of these lists of arguments is used to generate many lists of arguments. This setup allows built data structures to be reused, greatly reducing duplicated work [3].

As briefly touched upon, the python module receives the data set and queries from ANN-Benchmarks, and must expose at least three functions:

- `__init__`: receives metric and args from algos.yaml
- `fit`: receives the data set and should build the index to make it available in `query`
- `query`: receives a query point *q* and an integer *k* for returning the *k* nearest neighbor points to *q*

Additionally, for the eCP implementation the `set_query_arguments` function that receives additional parameters for querying has been included for the *b* parameter specified in algos.yaml. Lastly, the `__str__` function can be provided to plot the parameter settings for each run's data points when

[8] `https://github.com/erikbern/ann-benchmarks`

creating the website. This toString representation is not required but is a nice addition when evaluating the implementation using the interactive website, displaying the L and b parameters.

As ANN-Benchmarks currently support about 18 different algorithms each with their own dependencies, one can imagine the amount of complexity required to make them all function within the same environment. Before an algorithm can run, it must first have its Docker container built which is what the `install.py` script in ANN-Benchmarks does. After the docker image is built, the algorithms dockerfile is executed which should retrieve the algorithm repository and make sure it is available for the ANN-Benchmarks python module. For making continuous development easy the eCP docker file builds the wrapper inside the container, exposing it to the ANN-Benchmarks python module. When a new version of the algorithm is needed for testing, the docker container can simply be removed, deleting all dependencies and files within it, and `install.py` can be run again to create an updated eCP container. Although ANN-Benchmarks recommends using docker for comparing algorithms, it also supports a local mode that can be helpful during the development of an implementation.

The final configuration of the eCP algorithm inside the algos.yaml file can be seen on Listing 4

Listing 4 algos.yaml

```
eCP:
      disabled: false                          #should the algorithm run?
      docker-tag: ann-benchmarks-ecp           #docker file
      module: ann_benchmarks.algorithms.eCP    #python module
      constructor: eCP                         #constructor of module
      base-args: ["@metric"]
      run-groups:
        eCP:                                   #eCP run group
          args: [[1,2,3,4,5]]                  #L parameter
          query-args: [[1,2,3,4,5]]            #b for each L
```

To assist development, two utility python scripts have been written to streamline the process of wrapping the C++ code. The script `file_combine.py` file combines all source files into a single one and the result is then fed into the `interface_gen.py` file to 1) find all the signatures in the C++ source code and 2) generate the `eCP.i` interface file used by SWIG to create the wrapper.

These scripts are used in combination with several basic commands for moving files around, in `generate_wrapper.txt` that puts everything together, and creates the complete wrapper. This can be thought of as a 'pipeline' for automating the wrapping process. When the wrapper is created it is then possible to import it from python by importing the 'eCP_wrapper' module and using the `eCP_Index` and `query` functions as described.

3.6 Performance Improvements

Throughout development, the eCP implementation has been evaluated using the ANN-Benchmarks in an attempt to make query processing faster and improve recall. The first iteration of the eCP implementation merely had the functionality of the algorithm and used array lookups rather than pointers. From there, decisions on implementation details to improve performance were made and evaluated using ANN-Benchmarks. Throughout development, multi-threading and parallelization of queries has been kept out of the equation, since other projects on eCP have explored this topic [12, 5, 6]. Despite this, it could enable interesting approaches to the b parameter and batched queries in eCP. This could specifically be evaluated with ANN-Benchmarks batch mode [3].

In the early iterations of the eCP implementation the usage of std::vector was poor, given that it was used for fixed-sized lists. One example of this was representing descriptors as vectors although the length is fixed. This of course introduced a large overhead for points and was changed to an array representation with the length kept as a global variable which improved performance significantly. The `eCP_Index` and `query` function both make use of std::vector since SWIG by default treats it as a python list, and therefore handles the conversion implicitly. Due to this, we transform the received data set, in the form of a std::vector, into the point struct representation.

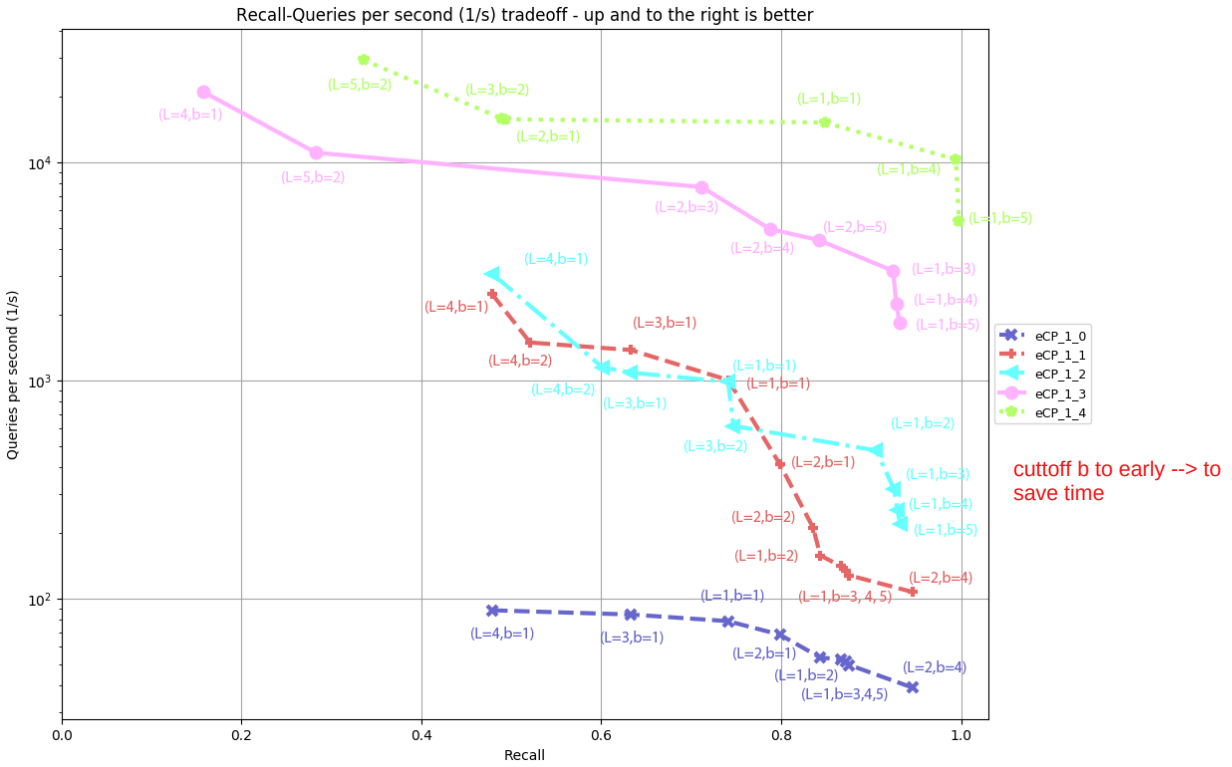

Figure 6. eCP development iterations)

Figure 6 shows the different implementation of eCP on the random-20 (table 1) set. Each graph is a release made after substantial changes. The large improvement in release 1.3 came from the introduction of pointers to arrays with length kept globally. Changes were also made to index structure in 1.3, resulting in different recall for the parameter settings compared to earlier releases

3.7 Code Readability

To have implementation be suitable for teaching purposes, an effort has been made towards making the code as readable as possible without sacrificing significant performance. The codebase has therefore been adhering to the ISO C++ coding standard [13] using a consistent naming scheme for files, variables, functions, and tests. An object-oriented style has been applied with regards to function signatures, no functions should return void as these functions contain side effects. Some cases have however been the exception being functions using an accumulator parameter when returning properly from the function was deemed infeasible. Classes have been structured with public functions at the top and private functions used inside them immediately below. The functions all have a high-level description, inside its header and implementation-specific details inside the implementation. As the implementation is meant to be a portable library, all functionality has been put into a single project exposing only the required functions and can be exported as a library allowing it to be used by importing the `eCP.hpp` header.

3.8 Summary

It is noticed that the usage of pointers increased in our various data structures as they became more abstract and with that, so did the risk of our program containing memory leaks. The profiling tool Valgrind[9] was actively used for profiling the memory usage of our implementation and indicating leaks. Valgrind was not a part of the wrapping pipeline created, but was used frequently throughout the development and could be added to ensure implementations with leaks would not be accepted by ANN-Benchmarks. Valgrind indicated that the final eCP implementation did not leak any memory as long as the index is deleted after usage. In the python module a SWIG object for the index is created which is collected by the python

[9] `https://valgrind.org/`

garbage collection. The final project structure with relevant files, is shown in Listing 5 with a general description of the most important files.

Listing 5 Project Structure

```
root folder
├── ann-benchmarks
│   ├── ann_benchmarks
│   │   ├── algorithms
│   │   │   └── eCP.py                    python module for ANN
│   ├── install
│   │   ├── Dockerfile.ecp                docker file for eCP
│   ├── algos.yaml                        algorithm configuration
├── eCP
│   ├── eCP
│   │   ├── data_structure.hpp            data structure implementation
│   │   ├── distance.cpp                  distance functions
│   │   ├── eCP.cpp                       ANN Benchmarks functionality
│   │   ├── eCP.hpp
│   │   ├── file_combine.py               combines source files
│   │   ├── pre-processing.cpp            index creation
│   │   ├── pre-processing.hpp
│   │   ├── query-processing.cpp          index querying
│   │   ├── query-processing.hpp
├── wrapper
│   ├── interface_gen.py                  generates eCP.i
│   ├── setup.py                          builds wrapper
├── generate_wrapper.txt                  pipeline
```

4 RESULTS

In this chapter the results of running the final version of the eCP implementation in ANN-Benchmarks is presented. First, the effect and trade-off between the L and b parameters are discussed. Then the results from smaller data sets and larger data sets provided by ANN-Benchmarks are compared to other state-of-the-art algorithms. Lastly, the performance of the euclidean- and angular-distance functions are compared when computing the real, and approximate distance.

All results have been generated on a Windows 10 machine with a 4 core Intel Core i7-4720HQ @ 2.60GHz CPU inside a virtual box running Ubuntu 18.04 with 4GB RAM. For running the largest data sets more than 4GB RAM was needed, and a second machine was therefore put to use. The largest data set results are not included in this report but can be found together with the machine specification in the GitHub repository[10]

Table 1 contains all data sets[11] used to evaluate the eCP implementation in this project. All data sets have been pre-split into train/test and ground truth data in the form of the top 10 or 100 neighbors. The ground truth data is for computing recall and ensuring that the index is not overfitting the training data.

Category	Dataset	Dimensions	Train Size	Test Size	k	Distance
Small	Random-20-E	20	9000	1000	10	Euclidean
	Random-100-E	100				
	Random-20-A*	20				Angular
Large	Fashion-MNIST	784	60,000	10,000	100	Euclidean
	SIFT*	128	1,000,000			
	GLOVE*	25	1,183,514			Angular

Table 1. The ANN-Benchmarks data sets eCP was evaluated on. Data sets marked with * have their results in the GitHub repository. The Random data sets are generated by ANN-Benchmarks

All graphs presented in this chapter have recall as the x-axis and queries per second on the y-axis. Recall, defined as the ratio of the number of points in the result set that are true nearest neighbors to the number k of true nearest neighbors, is computed using the formula presented in equation 7. Queries per second is the time it takes to retrieve the k nearest neighbors for a given query point and return it as a result set. The y-axis is log-scaled to show and be able to compare the wide range of queries per second obtained from the different ANN algorithms. When looking at the comparisons of eCP to other state-of-the-art ANN algorithms, it is important to note that the other algorithms have been run with parameter settings from the ANN-Benchmarks repository[12].

4.1 Parameter Settings

As seen in figure 7 where indexes of different L are queried with different b values, incrementing L improves retrieval speed for query processing. This is to be expected since points are effectively being "pruned" at search time in the larger index which results in fewer comparisons. Naturally, increasing L also results in more clusters of smaller size, which in turn reduces recall unless b is tuned accordingly. On the contrary, b improves recall while slowing down query processing. This is also to be expected as more comparisons have to be made during traversal and scanning of the b nearest cluster's points. This trade-off is unique to the data set, and therefore experiments has to be made to decide on an L parameter suitable for the eCP index.

4.2 Data Set Sizes

Experiments on the data sets included in ANN-Benchmarks generally showed that $L = 1$ was suitable for ensuring a recall near or above 0.6 for most data sets. Therefore the following figures have $L = 1$, and increasing b values to obtain $recall = 1$.

From figure 8, it is seen that the eCP implementation performs decent on smaller data sets. As b increases, queries per second only slightly decreases as the recall approaches 1. Here eCP can compete

[10] `https://github.com/nimertz/eCP`
[11] `https://github.com/erikbern/ann-benchmarks/#data-sets`
[12] `algos.yaml` in `https://github.com/erikbern/ann-benchmarks/`

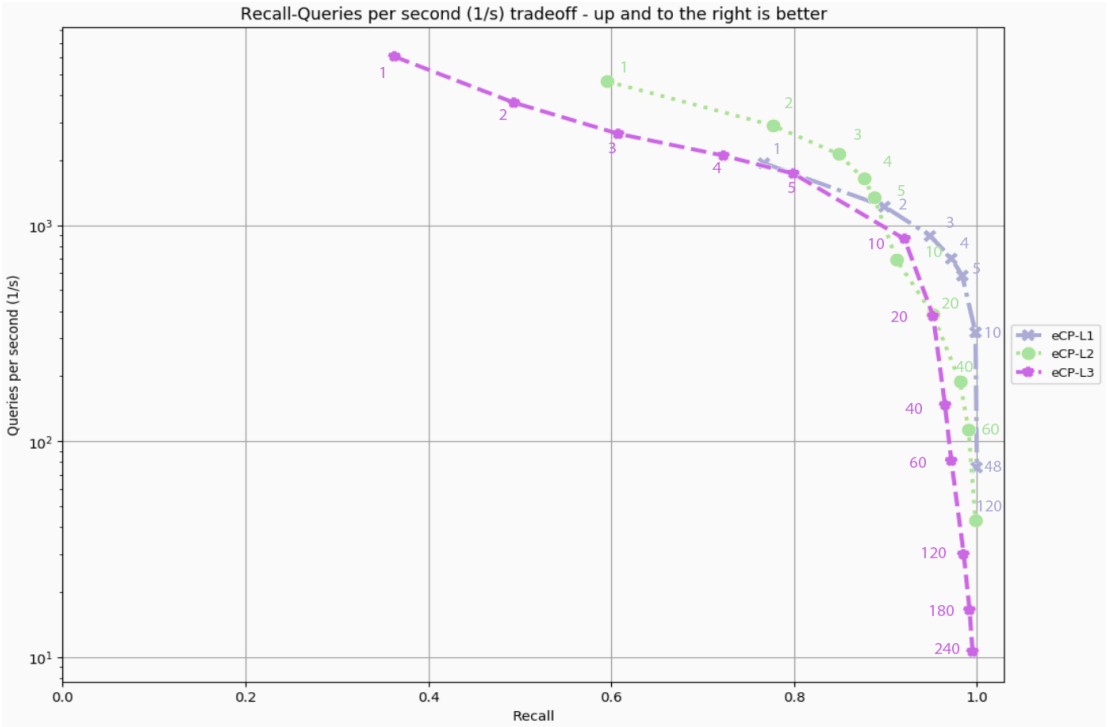

Figure 7. Different L parameters for Random-100-E with b values increasing to obtain full recall

with and surpass algorithms such as the tree-based Annoy[13] and graph-based PyNNDescent[14].

Looking at figure 9, it is observed that the eCP implementation performs significantly worse on the larger high-dimensional data sets included in ANN-Benchmarks. Here the other algorithms significantly surpass its retrieval speed.

4.3 Distance Function Performance

Timing ten million iterations of each distance calculation[15] with points of 784 dimensions five times and averaging the time, results in table 2. It is seen that angular distance runs about 30% slower than the euclidean distance function, which should be expected considering angular distance performs heavier calculations. These calculations are based on the real distances (equations 1 and 2) and approximations (equations 9 and 10) where some computations are left out. This highlights that even though a certain distance function might best describe the similarity of data, it can come at a cost to performance regarding index construction and query time. Therefore, the trade-off between calculation cost and precision when it comes to similarity is essential to consider when working with high-dimensional data. The approximation is seen to only speed up the calculations by approximately half a percent which makes the trade-off between the real distance and the approximation questionable

Distance function	Real distance (seconds)	Approx. distance (seconds)	Improvement
Angular	$\approx$ 68,63	$\approx$ 68,35	$\approx$ 0,40 %
Euclidean	$\approx$ 43,80	$\approx$ 43,57	$\approx$ 0,52 %

Table 2. Results of comparing calculations of real and approximate distance

[13] `https://github.com/spotify/annoy`
[14] `https://github.com/lmcinnes/pynndescent`
[15] Run on computer with Intel Core i5-8600K CPU @ 3.60GHz, 3600 Mhz, 6 Core(s), 6 Logical processor(s), 16GB RAM

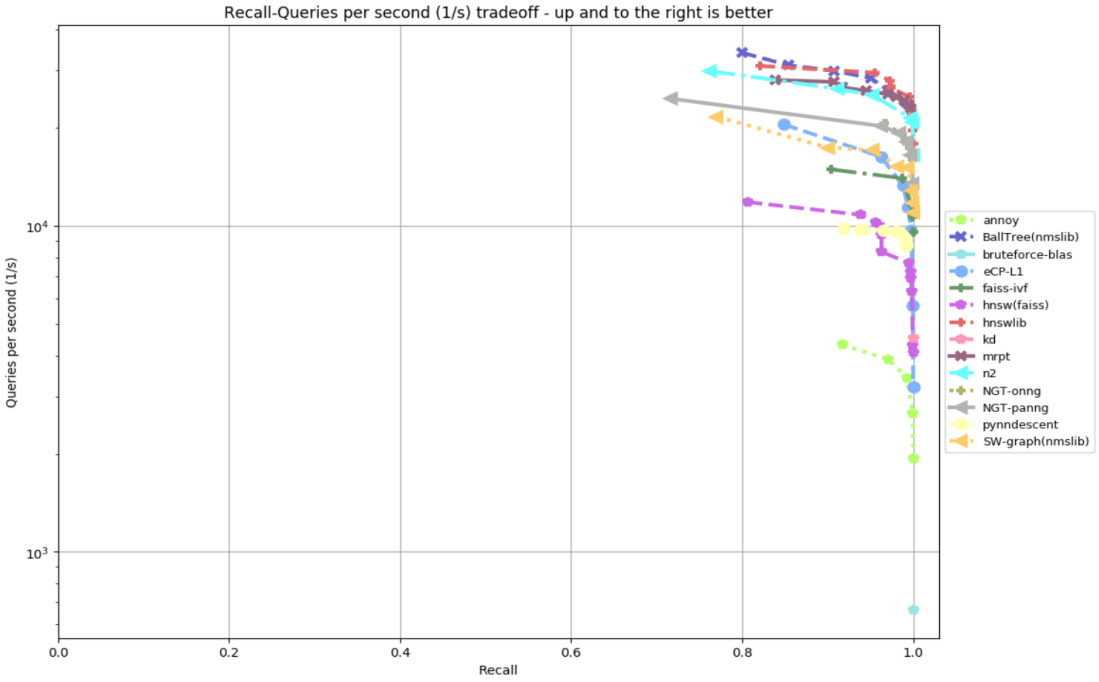

Figure 8. Result for $L = 1$ on Random-20-E with b values ranging from 1-5 and a b sure to obtain full recall

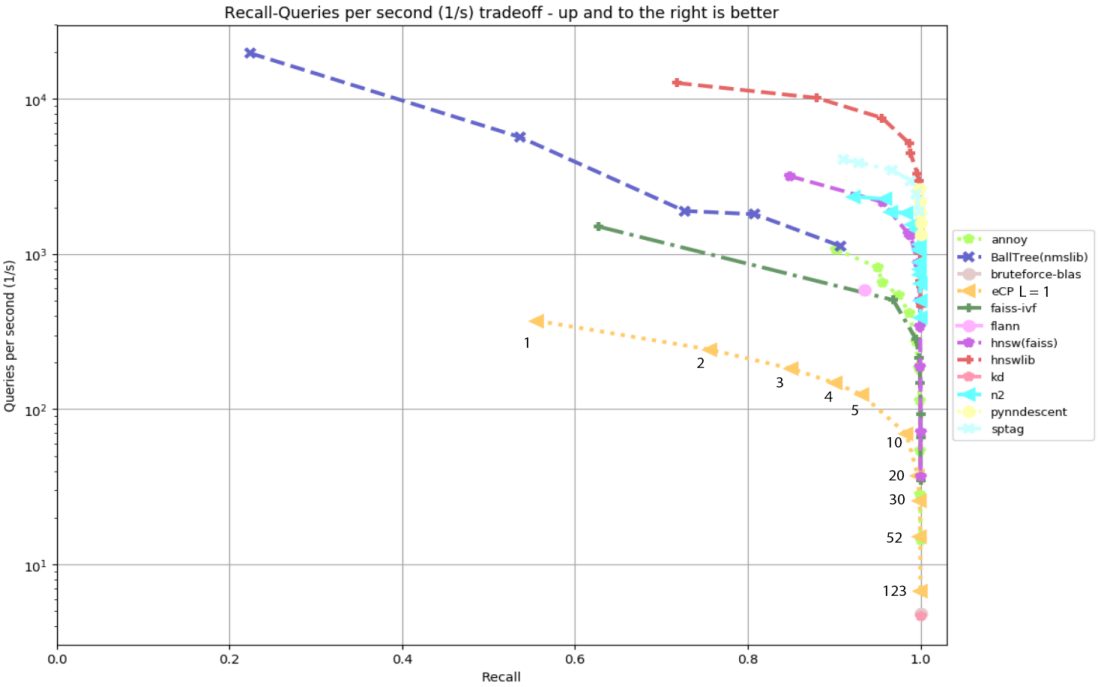

Figure 9. Results for Fashion-MNIST with $L = 1$ and b increasing for obtaining better recall

5 CONCLUSION

5.1 Current Issues and Future Work

There are definitely opportunities for improving the retrieval performance of the eCP implementation by studying and improving cache performance. Doing this of course, comes with a caveat, as it is likely to reduce readability. As ANN-Benchmarks support a batch-mode, it could be interesting to explore how eCP would perform using parallelism for batch queries, where the index would receive a sequence of queries all at once and return results for all of them in that sequence. The distance calculation results proved that is was not worth sacrificing the real distance for an approximation in this case. Experimenting with a cutoff-point, at which the calculations stops if it is already known that the distance will be worse, or to experiment with the loop unrolling technique[16] could be interesting to explore.

In the implementation the cluster leaders are chosen as representatives, which in turn drastically speeds up index creation in the pre-processing phase. Having said this, Chierichetti et al. suggestion of using centroids could be implemented and evaluated using ANN-Benchmarks to see if this is the better choice for in-memory performance. This is likely, since clusters will have a higher chance of representing the points they contain more accurately. Of course this will end up being a trade-off between query-processing and pre-processing time. The choice of representatives was made based on the assumption that the ANN-Benchmarks data sets have no particular sorted order. If this is not the case the index creation will be skewed. A solution to this problem would to be do a random sampling of the data set instead of taking the first elements of the data set. This would be a priority for future work, as the result in chapter 4 indicated better performance on the randomly generated data sets.

In terms of cluster sizes, more experiments could be made to reduce memory usage. Specifically, a formula like the one presented in equation 8 could be explored based on the parameter settings and data set size to reserve the average memory of a cluster. A traversal of the index could be made to shrink clusters to reduce memory usage in our implementation, and could be further explored using ANN-Benchmarks to evaluate the index size and creation time.

In chapter 4 the default parameter settings for other ANN algorithms from the ANN-Benchmarks repository was used. For more accurate and fair comparisons to other state-of-the-art ANN algorithms in terms of what algorithm is the best choice for a certain data set, it would be necessary to look more into their respective parameters.

Looking at other algorithms[17] in ANN-Benchmarks, it is seen that they normalize the input data set before pre-processing. It may therefore, be worthwhile to look into, as it would likely improve the performance of eCP given an index that fits the data set better.

ANN algorithms continue to be relevant and the ANN benchmarking tool has proven itself to be a reliable and usable tool for comparing these types of algorithms. It makes it easy to compare new experimental algorithms to already established algorithms in a visual way that is both easy to understand and highly interactive. Looking through the documentation, it is seen that the target of this application is not bachelor students, but more likely researchers or students with a masters degree in computer science. Should this tool be used in educating students at a bachelor level, we strongly recommend a more thorough walk-through of installation and a complete example of how the tool is used, along with an overview of python file parameters. This could be done starting with a well-known sample algorithm and describing the procedure from embedding to evaluating.

Since SWIG is an automatic wrapping tool where you get a lot for free, it most likely sacrifices performance for flexibility compared to manually creating a wrapper tailored to the implementation. It may therefore, be beneficial to explore manually building the wrapper to have complete control over the mappings in C++ and python.

5.2 Contributions

We have developed a C++ implementation of the extended cluster pruning algorithm and compared it to other state-of-the-art ANN algorithms using ANN-Benchmarks. As seen in chapter 4, eCP performs especially well on the smaller data sets included in ANN-Benchmarks and can compete with some of the state-of-the-art-algorithms. The implementations retrieval performance on the larger data sets was weaker than expected based on Chierichetti et al. and Guðmundsson et al. [10, 4] findings, but the recall is as expected due to the approximate nature of eCP. As described in chapter 3, the eCP implementation

[16] `https://en.wikipedia.org/wiki/Loop_unrolling`

[17] balltree, bruteforce, faiss, flann, kdtree, ishf, mprt, nearpy

has been created with an emphasis on readability for allowing the code to be used in teaching. This has resulted in a C++ implementation that should allow computer science students to get familiar with the eCP algorithm and create improved versions without having to start from scratch.

REFERENCES

[1] Comprehensive guide to approximate nearest neighbors algorithms. towardsdatascience.com, 2011. URL https://towardsdatascience.com/comprehensive-guide-to-approximate-nearest-neighbors-algorithms-8b94f057d6b6.

[2] Richard Ernest Bellman. *Dynamic programming*. Princeton University Press, 1957. ISBN 978-0-691-07951-6.

[3] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. A benchmarking tool for approximate nearest neighbor algorithms. *ANN-Benchmarks*, 2018.

[4] Gylfi Þór Guðmundsson, Björn Þór Jónsson, and Laurent Amsaleg. A large-scale performance study of cluster-based high-dimensional indexing. *Cluster-Based High-Dimensional Indexing*, 2010.

[5] Gylfi Þór Guðmundsson, Laurent Amsaleg, and Björn Þór Jónsson. Distributed high-dimensional index creation using hadoop, hdfs and c++. *Tenth International Workshop on Content-Based Multimedia Indexing (CBMI)*, 2012.

[6] Gylfi Þór Gudmundsson Laurent Amsaleg Diana Moise, Denis Shestakov. Indexing and searching 100m images with map-reduce. *ACM International Conference on Multimedia Retrieval (ICMR)*, 2013.

[7] Björn Þór Jónsson Michael J. Franklin Gylfi Þór Guðmundsson, Laurent Amsaleg. Towards engineering a web-scale multimedia service: A case study using spark. *ACM Multimedia Systems Conference (MMSys)*, 2017.

[8] Cui Yu (eds.). *High-Dimensional Indexing: Transformational Approaches to High-Dimensional Range and Similarity Searches*. Springer-Verlag Berlin Heidelberg, 1 edition, 2003. ISBN 978-3-540-44199-1,978-3-540-45770-1.

[9] Björn Þór Jónsson Laurent Amsaleg Herwig Lejsek, Friðrik Heiðar Ásmundsson. Nv-tree: An efficient disk-based index for approximate search in very large high-dimensional collections. *IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE, VOL. 31, NO. 5*, 2009.

[10] Flavio Chierichetti. Finding near neighbours through cluster pruning. *Finding Near Neighbours Through Cluster Pruning*, 2007.

[11] 6 tips to supercharge c++11 vector performance. acodersjourney.com, 2016. URL https://www.acodersjourney.com/6-tips-supercharge-cpp-11-vector-performance/.

[12] Andri Mar Björgvinsson. Distributed cluster pruning in hadoop. *Distributed Cluster Pruning in Hadoop*, pages 1–48, 2010.

[13] Coding standards c++. isocpp.org. URL https://isocpp.org/wiki/faq/coding-standards.