CGL

Computational Geometric Learning

Orthogonal Range Clustering

Ebrahim Ehsanfar Dan Feldman Christian Sohler

CGL Technical Report No.: CGL-TR-73

Part of deliverable:	WP-I/RTD
Site:	TUDO
Month:	36

Project co-funded by the European Commission within FP7 (2010–2013)
under contract nr. IST-25582

1 Abstract

In this report, we explain the general idea and the notation of a fast technique that lead to an approximation for orthogonal range clustering using coreset constructions. Given a range space $(X, \Re)$ and a set $P \in X$ of n points in d-space, the goal of orthogonal range clustering is to find a $k-$clustering C of the points within any arbitrary range $Q \in \Re$. For computing an approximation for C, we construct a weighted summary S of the points within Q by preprocessing the points into a data structure. We then show that the cost of clustering based on the summary has only a small different with the cost of the clustering for the original point set. For yielding summary S in a canonical approach, we modify the data structure by small coreset constructions namely coreset plug-ins. Then for any range query Q, we analyse the process to compute an approximate clustering using this modified data structure.

2 Introduction

For defining the problem, consider a database for bank record transactions. In such a database the information like name of account owners, date of transaction, amount of money, receiver account number etc, of each transaction are stored. A typical query is e.g. to find all transactions such that the amount is between 1000 and 2000 Euro and that happened between 10:40 AM and 11:20 AM. To formulate this problem geometrically, we show each transaction by a point in the plane. The x-coordinate of the point is the time of transaction, and the y-coordinate is the amount of money. We want to report all the points inside an axis-parallel query rectangle. To answer such query we have to report all points P inside the axis-parallel box Q. Such a query is called an *orthogonal range query*. In a range query we can provide some information about the points P lying within Q, e.g. to return the list of all points P that lie within Q (range reporting) or to return the number of all points P that locate within Q (range counting). These problems have been well studied and there are several approaches to solve them properly. The problem that is interesting for us in this work is to cluster the points within Q. The standard approach for such clustering is to first report the points with a currently know method then apply a clustering approach on the reported points. In this survey we present a direct mechanism to find a clustering of the points inside the rectangular range Q without needing to report all of them in advance. Section 3 describes range spaces properties to formulate the problem officially. In Section 4, we explain the data structures used for range reporting problem. Section 5 contains different concepts regarding k-means clustering. In Section 6 we presents our modified data structure for orthogonal range clustering. Section 7 includes the experimental results.

3 Range Spaces Properties

For modelling the problem, an important fact is that the subsets of ranges for a point set in the space (e.g. rectangles) are usually smaller than the set of all possible subsets of them. A range space is defined by a pair $(X, \Re)$ that X is a point set and $\Re$ is a subset of the power set of X (all possible subsets of X). For the set $P \subseteq X$, define $\Pi_R(P) = \{P \cap Q | Q \in R\}$ as the collection of subsets of P that are the intersection of P and a range Q in the range space. We use this notation from [13] and [8] in the rest of this paper. A typical way to solving almost all range queries is to represent reported points P as a set of *canonical subsets* $\{P_1, P_2, ..., P_k\}$ $(P_i \subseteq P)$, such that any reported set of points can be formed as a union of these canonical subsets. Many ways to select canonical subsets have been given with different time

and space complexities. We will discuss some of them in this work. A practical way to define canonical subsets is by using a range tree. We start by solving the 1-dimension range queries. Suppose a given set $P = \{p_1, p_2, ..., p_n\}$ of points on a real line and an interval $[u, v]$ and the problem is to report all the points within the interval. One can put the sorted points set P in the leaves of a binary search tree. We can define $O(n)$ canonical subset associated with any subtree of this binary tree (or with more precision, with a node of this binary tree that is rooting a subtree). Any range Q can be identified in $O(\log n)$ time from this structure. For reporting queries, since the leaves of each subtree can be listed in time that is proportional to the number of leaves in the tree (a basic fact about binary trees), it follows that the total time in the search is $O(\log(n) + k)$, where k is the number of reported points. In the following we describe the ways for solving range searching problem in higher dimensional spaces by reviewing the currently know data structures.

4 Data Structures for Range Searching

We have seen that 1-dimensional range reporting queries can be answered in $O(\log n)+k$ time, using $O(n)$ storage. The problem of finding maximal subtrees that are lying within the range is fundamental to all range search data structures. The only question is how to organize the tree and how to locate the desired sets. Here we study some data structures for orthogonal range reporting problem. Bentley et al. [7], Agarwal et al. [2] and Shi et al. [25] review several data structures for d-dimensional range searching. One of the most popular data structures to solve the range searching problem is the kd-tree [6] which is a binary search tree that stores the points in a d-dimensional space. At any intermediate node, the kd-tree partition the d-dimensional space in two parts by a $(d-1)$-dimensional hyperplane. In each level, the partitioning hyperplane is chosen based on one of d possible coordinates. There are several versions of kd-trees, e.g. an improved version in [16] which is the adaptive kd-tree. During splitting, the adaptive kd-tree chooses a hyperplane that divides the space in two sub-spaces with equal number of points. Devroye et al. [14] studied range search on squarish kd-trees and random kd-trees [10]. The kd-B-tree [22] uses properties of both the adaptive kd-tree and the B-tree [12]. The B-tree is a generalization of a binary search tree in which a node can have more than two children, in fact a B-tree of order m is a balanced tree that each node contains at most $m-1$ and at least $\lceil \frac{m}{2} \rceil - 1$ elements and any intermediate node contains i elements and has $i+1$ children within some pre-defined range. In [26] range searching using B-trees and their generalizations is well described. The R-tree ("R" is for rectangle) [17] is a hierarchical data structure that is derived from the B-tree. It is a balanced search tree in which one groups nearby objects and represent them with their minimum bounding rectangle in the next higher level of the tree. Since all objects locate within this bounding rectangle, a query that does not intersect the bounding rectangle also cannot intersect any of the contained objects. The range searching mechanism in R-trees is discussed in [20]. A packed R-tree is studed by Roussopoulos and Leifker [23] is an R-tree which is built by applying a nearest neighbour property to group objects in a node after the set of objects has been sorted based on a spatial rule. Another version of the R-tree is the R^+-tree [24]. The idea for designing R^+-tree is to avoid overlap of the bounding rectangles. Another generalization is presented by Beckmann et al. [5] to work on an improved version of the R-tree, the R^*-tree, by minimizing the overlap region between sibling nodes in the tree. Any mentioned generalization of R-tree can be constructed in time $O(m)$ with space complexity $O(m^2)$ for m rectangles. Although the Priority

	Complexities	
Data Structures	Space	Time
kd-trees	$O(n)$	$O(n^{1-1/d}+k)$
Range tree	$O(n\log^{d-1} n)$	$O(\log^d n+k)$
Overlapping k-ranges	$O(n^{1+\epsilon})$	$O(\log n+k)$
Non-overlapping k-ranges	$O(n)$	$O(n^{\epsilon+k})$
R-tree	$O(n)$	$O(n+k)$
Priority R-tree	$O(n)$	$O(n^{1-1/d}+k)$

Table 1: Complexities of different range searching data structures

R-tree, or PR-tree is presented by Lars Arge et al. [3] is significantly better than other R-tree variants regarding the time query. We compare complexities of the some data structures in Table 1.

5 Clustering and Approximate Methods

k-means and k-median clustering are the methods of cluster analysis which aims to partition n points into k clusters in which each point belongs to the cluster with the nearest center. The first and simplest algorithm for k-clustering is Lloyd's algorithm [21]. It tries to minimize the within-cluster sum of distances. The algorithm proceeds as follows:

1. Draw k random center from the input and assigne any point to the nearest center and partition the data to k set.
2. Calculate the centroid of each set.
3. Assign each point to the set corresponding to the closest centroid.
4. Repeat the last two steps until nothing is moved around, or until some maximum number of iterations has been reached.

The algorithm repeats until the centroids do not change or when the error reaches a threshold value. The computational complexity of algorithm is $O(NKd)$.

For the rest of this section, we define the k-median and k-means problem more formally.

Definitions 1. *Suppose $d(x,y)=\|x-y\|$ is the Euclidean distance of $x,y\in R^d$. We use $d(x,C)=min_{c\in C}d(x,c)$ and $cost(P,C)=\Sigma_{x\in P}d(x,C)$ for any $P,C\subset R^d$ and for a weighted subset S with a positive weight function $w:s\Rightarrow R^+$, $cost_w(S,C)=\Sigma_{y\in S}w(y)d(y,C)$. Now given an input set $P\subset R^d$ with $|P|=n$ and an integer number k, k-median clustering problem is to find a set $C\subset R^d$ with $|C|=k$ that minimizes $cost(P,C)$ (that is named clustering cost of P based on C). By defining $d^2(x,y)=\|x-y\|^2$ (the squared Euclidean distance) metric to the mentioned notation, we can define k-means clustering problem in a same way.*

5.1 Faster Algorithms

There are lots of generations of Lloyds algorithm that can act faster to get a good result for the k-means problem. A negative point for Lloyds algorithm is the fact that it is very sensitive on its initial values. The better initial values can help us to

Algorithm: k-means++ seeding procedure
$C \leftarrow \{x_i\}$ where x_i is uniformly chosen **for** $j \in [k]$ **do**
Pick node x with probability proportional to $\frac{d^2(p,C)}{cost(P,C)}$
Add x to C
end **for**
return: C

get the results very faster (or even with smaller error). Some of those algorithms named *seeding* algorithms which give initial assignments to Lloyds algorithm. A well known seeding procedure for k-means is called k-means++ [4]. In each iteration, the next center is chosen randomly from the input points but the distribution over the points is not uniform. Each point chosen with probability proportional to the minimal square distance from it to a currently picked center. More formally suppose point set S already picked from P. Then, the next element $p \in P$ of S chosen with probability $\frac{d^2(p,S)}{cost(P,S)}$. We say S is chosen at random according to d^2. This approach gives an $O(\log(k))$ approximation guarantee.

5.2 Coreset Constructions for k-clustering

The idea of coresets is to compute a weighted point set which we call a (k, ϵ)-coreset. For an optimization problem, a coreset is a subset of the input, such that we can get a good approximation to the original input by solving the optimization problem directly on the coreset. For this, to get a proper approximation, one needs to compute a coreset, as small as possible from the input, and then solve the problem on the coreset using known techniques. Many coreset constructions have been introduced for geometric approximation and also there are lots of coreset constructions that have been used for k-means clustering, most notably [9]. This construction is used for clustering problem for the first time or [18] where the size of the coreset is independent of the size of the input set. Chen [11] present a coreset costruction with size linear in d. Thus, these coresets are good for high-dimensional point sets. Feldman et. al [15] have given a coreset constructions which its size is independent of n and d. A formal definition of a coreset for k-clustering is as follow:

Definitions 2. *Let $k \in N$ and $\epsilon \leq 1$. A weighted multiset $S \subset R^d$ with positive weight function $w : s \Rightarrow R^+$ and $\Sigma_{y \in S} w(p) = |P|$ is called a (k, ϵ)-coreset of P if and only if for any $C \in R^d$ of size $|C| = k$ we have:*

$$(1 - \epsilon)cost(P, C) \leq cost_w(S, C) \leq (1 + \epsilon)cost(P, C) \tag{1}$$

In the next sections, we use the coreset construction to modify data structures for orthogonal range clustering.

5.3 Coreset Plugin

The following observation is mentioning an interesting property of coresets which is useful in our methods.

Observation 1. *[19] Suppose that $P \in R^d$ and disjoint union $P = \bigcup_{i=1}^{m} P_i$ and C_i is a (k, ϵ)-coreset for P_i, then $C = \bigcup_{i=1}^{m} C_i$ is a (k, ϵ)-coreset for P.*

More precisely, if a data set is split into k subsets, and we compute a $(1 + \epsilon)$-coreset of size s for every subset, then the union of the k coresets is a $(1+\epsilon)-$coreset of the whole data set and has size $k.s$. So a reasonable idea is to partition the dataset

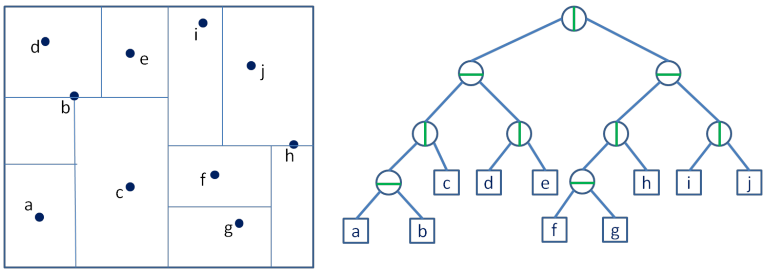


Figure 1: A simple kd-tree in 2D with the associated space partitioning

into groups, compute a coreset for each group and find the union of the resulting coresets. This is also especially helpful if the data is already given in a distributed fashion on k. The small coresets can then be reported to approximately solve the clustering problem.

As it was mentioned before, in range reporting problem, the better way is to form point set P as a set of canonical subsets $\{P_1, P_2, ..., P_k\}$ ($P_i \subseteq P$), such that any reported set of points can be shown as an union of these canonical subsets. Then by having a coreset for any of canocical subsets, one can compute the union of the coresets of the points within the range as the union of the canonical coresets. One way to provide the canonical coresets is to compute and store a coreset associated with any of the canonical subsets as a coreset plugin. Then during the range query, it is enough to reach the coreset plugin of any canonical subset within the range and collect them to form a coreset union for all the points within the range. In the next section, we study an enriched version of a kd-tree with a coreset plugin to provide a fast technique for approximating clustering problem for range queries.

6 Coreset Kd-trees

Kd-tree is a kind of a partition tree and is a well known data structure for range searching especially in high dimensional spaces. It is useful for many searching problems (e.g. nearest neighbour searching). In this data structure, points are stored at the leaves. In dD for any node, we divide space by splitting along the i-coordinates. Any internal node of the tree is associated with the *cutting dimension*, the *cutting value* and can be enrich by the number of points of its subtree. note that In R^d, the cutting dimension can be represented from 0 to $d-1$. For cutting dimension i, all points with $i'th$ coordinate less than or equal to the current cutting value are stored in the left subtree and the remaining points are stored in the right subtree (Fig. 1). We do this procedure until only a single point remains that we store in a leaf node.

Each node of the tree is associated with a rectangular region of space (called a cell). With the fact that child's cell is contained inside the parent's cell, these cells define a hierarchical partitioning for the space (Fig. 1). The main factor in constructing a kd-tree is to turn through the dimensions one by one. To guarantee that the tree has height $O(\log n)$, the best method is to let the cutting value be the median coordinate along the cutting dimension, however one can introduce another splitting function for partitioning the space for generating new nodes.

Proposition 1 ([8])**.** *Given n points, it is possible to build a kd-tree of height $O(\log n)$ and space $O(n)$ in time $O(n \log n)$ time.*

It is possible to build a kd-tree in $O(n \log n)$ time by a simple top-down recursive procedure. It is needed only to calculate the median of the points in each step. One

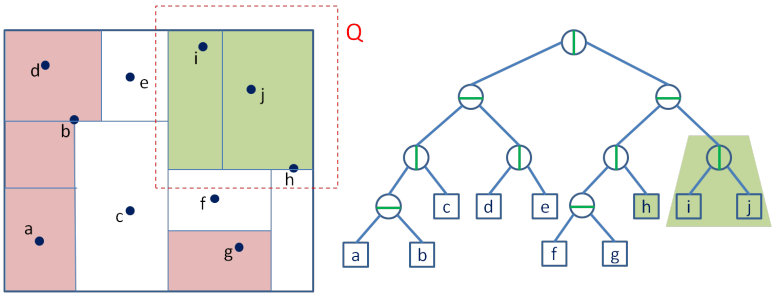


Figure 2: Range query in a kd-tree

way to do this is to maintain two lists of pointers to the points, one sorted by i-coordinate and the other containing pointers to the points sorted according to their i+1-coordinates. Using these lists, it is an easy matter to find the median at each step immediately. For computing the construction complexities, suppose $x = s$ is the cutting value, then all points with $p_x \leq s$ go into one list and those with $p_x > s$ go into the other. In R^d it takes $O(d)$ time per point). This leads to a recurrence of the form $T(n) = 2T(n/2) + n$, which solves to $O(n \log n)$. As there are n leaves and each internal node has two children, it follows that the number of internal nodes is $n - 1$ and the total space requirements are $O(n)$.

To enrich a kd-tree with a coreset construction, the procedure is very similar to the range-trees. Suppose $P_Q = \{P_1, P_2, ..., P_h\}$ is the set of reported canonical subsets of queried range Q on the tree. We are looking to compute a coreset for any of canonical subtrees of the kd-tree using an *adaptive sampling according to* D^2. This coreset construction is from [1] which is based on the idea of the k-means++ seeding procedure for k-means problem. In this method, for a coreset of size m, the construction is as follows. First, we pick up a set $S = \{q_1, q_2, ..., q_m\}$ form the point set P at random according to D^2. Suppose P_i is the points from P which are closest to q_i. We make the weighted set S as our coreset by using a weight function $w : S \to R$ with $w(q_i) = |P_i|$. For evaluating the efficiency of this construction we have the following proposition:

Proposition 2. *[1] If S is a multiset of $m = \Theta(\frac{k \log n}{\delta^{d/2} \epsilon^d} \log^{d/2} \frac{k \log n}{\delta^{d/2}})$ sampled points according to D^2 from a set P, then with probability at least $1 - \delta$ the weighted multiset S is a $(k, 6\epsilon)$-coreset of P.*

Now for checking the kd-tree performance, let Q be a given range, and u be the current node in the kd-tree. We assume that each node u is associated with its rectangular cell $u.cell$.

The kd-tree search algorithm traverses the tree recursively. If it arrives at a leaf cell, we check to see whether the associated point, $u.point$, lies within Q in $O(1)$ time. Else, u is an internal node. If $u.cell$ is disjoint from Q, then we know that no point in the subtree of u is in the query range, and so there is nothing here useful for us. If $u.cell$ is entirely contained in Q, then all the points in this subtree (a canonical subset) should be taken into account. Else, node u has only a little overlap with Q. In this case we recurs on two children of u. Then we only report the associated coreset to any canonical subtree rooted at the points within the range query.

For analysing the query time we need to find the number of nodes visited by this method.It is shown that the total number of visited nodes in 2D is $O(\sqrt{n})$ [8]. We say that a node is *expanded* if it is visited and both its children are visited by

the recursive range count algorithm. A node is expanded if and only if the cell overlaps the range without being contained within the range. We say that such a cell is *stabbed* by the query. To bound the total number of nodes that are expanded in the search, it suffices to bound the number of nodes whose cells are stabbed. For dD also it can be shown that the query time is bounded by $O(n^{1-\frac{1}{d}}+K)$ [8] which K is the number of reported points.

Lemma 1. *Let $k \in N$ and $0 < \epsilon \leq 1$ be a precision parameter and $0 < \delta \leq 1$ be an error probability, given a set P of n points, there exist a coreset kd-tree of size $O(n)$ so that for any dD orthogonal range R with probability of at least $1-\delta$ we can find a $(k, 6\epsilon)$-coreset of size $O(n^{1-\frac{1}{d}} \cdot (\frac{d}{\delta\epsilon})^{O(d)} \cdot k \cdot \log(n) \cdot \log(\frac{k \cdot \log(n)}{\delta\epsilon}))$ of P in time $O(n^{1-\frac{1}{d}} \cdot (\frac{d}{\delta\epsilon})^{O(d)} \cdot k \cdot \log(n) \cdot \log(\frac{k \cdot \log(n)}{\delta\epsilon}))$.*

Proof. As we mentioned before, we construct a kd-tree using a $(k, 6\epsilon)$-coreset construction of size

$$m = (\frac{d}{\delta\epsilon})^{O(d)} \cdot k \cdot \log(n) \cdot \log(\frac{k \cdot \log(n)}{\delta\epsilon})$$

so that we build a kd-tree in a canonical way but store a coreset plugin of any canonical sub-trees. It is enough to choose a split point which halve the number of points in each level. The space required to build the tree in level i is (root is in level 0):

$$S_i = 2^i \cdot (\frac{d}{\delta\epsilon})^{O(d)} \cdot k \cdot \log(\frac{n}{2^i}) \cdot \log(\frac{k \cdot \log(\frac{n}{2^i})}{\delta\epsilon})$$

So in general the size of the coreset kd-tree is:

$$\Sigma_{i=0}^{\log(n)} S_i = \Sigma_{i=0}^{\log(n)} 2^i \cdot (\frac{d}{\delta\epsilon})^{O(d)} \cdot k \cdot \log(\frac{n}{2^i}) \cdot \log(\frac{k \cdot \log(\frac{n}{2^i})}{\delta\epsilon})$$

Let

$$M = \Sigma_{i=0}^{\log(n)} 2^i \cdot \log(\frac{n}{2^i}) \cdot \log(\frac{k \cdot \log(\frac{n}{2^i})}{\delta\epsilon})$$

$$= \Sigma_{i=0}^{\log(n)} 2^i \cdot \log(\frac{n}{2^i})(\log(\frac{k}{\delta\epsilon}) + \log(\log(\frac{n}{2^i})))$$

$$= \Sigma_{i=0}^{\log(n)} 2^i \cdot \log(\frac{n}{2^i}) \log(\frac{k}{\delta\epsilon}) + \Sigma_{i=0}^{\log(n)} 2^i \cdot \log(\frac{n}{2^i}) \log(\frac{n}{2^i})$$

$$= O(n \log(\frac{k}{\delta\epsilon})) + \Sigma_{i=0}^{\log(n)} 2^i \cdot \log(n-i) \log(\log(n-i))$$

set $j = \log(n) - i$, so we can write the second part as follow:

$$\Sigma_{j=0}^{\log(n)} 2^{(\log(n)-j)} \cdot j \cdot \log(j)$$

$$= n\Sigma_{j=0}^{\log(n)} 2^{-j} \cdot j \cdot \log(j) < n\Sigma_{j=0}^{\log(n)} 2^{-j} \cdot j^2$$

$$\leq n \cdot 2^{-\log n}(-\log^2(n) - 4 \cdot \log(n) + 6 \cdot (2^{\log n} - 1)) \leq 6n$$

Note that $\Sigma_{i=0}^{\log(n)} 2^i \cdot \log(n-i) \log(\log(n-i))$ is $\Omega(n)$, so the final result is tight:

$$\Sigma_{i=0}^{\log(n)} 2^i \cdot (\frac{d}{\delta\epsilon})^{O(d)} \cdot k \cdot \log(\frac{n}{2^i}) \cdot \log(\frac{k \cdot \log(\frac{n}{2^i})}{\delta\epsilon}) = O((\frac{d}{\delta\epsilon})^{O(d)} \cdot n \log(\frac{k}{\delta\epsilon})). \quad (2)$$

For computing the query time based on this data structure, we already know that the query visits $O(n^{1-\frac{1}{d}})$ nodes and theresfore the overal query time is $O(n^{1-\frac{1}{d}}+K)$

in which K is the number of reported points (the coreset size in this case). It is straightforward to see that during the range query for any visited point, one can report at the most one canonical subtree (i.e. the corresponding coreset plugin) as at the most one of the children of the current node should be reported but not both. So the overal size of the reported coreset is bounded by $O(n^{1-\frac{1}{d}} \cdot (\frac{d}{\delta\epsilon})^{O(d)} \cdot k \cdot \log(n) \cdot \log(\frac{k \cdot \log(n)}{\delta\epsilon}))$ and therefore the query time is the same.

Hence, the only thing left to do is to prove that the cumulative error of coreset construction is bounded. Each canonical coreset plugin i has a failure probability δ denote by $Pr(X_i)$, and we have less than n plugins in the tree (binary tree has $n-1$ inner nodes and reported leaves of the tree are singletons which represent themselves in the final union coreset with error 0) so by Boole's inequality $S = \cup X_i$ holds with $\delta' = n \cdot n$

we have:

$$Pr(\cup X_i) < \Sigma_i Pr(X_i) < n.\delta$$

so it is enough to set $\delta < \frac{1}{n}$.

□

7 Experimental Results

We performed our experiments on a machine with 3.0 GHz dual core Pentium with 4 GB main memory, using windows vista. We have done our experiments on some random and real data set. A description of the datasets can be found in the next subsection. We compared our approach for orthogonal range clustering using modified kd-tree with the standard methods that are described in this survey. In summary the procedure is to modify a coreset kd-tree for datasets and query a range to reach a coreset of points within the range and then to apply k-means++ (as a fast clustering technique) on the reported points where the standard way is constructing a normal data structure on the points, and after completing the range query, to apply k-means++ on all of the reported points within the range. We run the implementation for two range queries. For the first time, we make a query for 25% of the points range. The second experiment run based on a 90% range query. We repeat the implementation for each range three times and present the average amount for the results. The results demonstrate that our data structure is competitive with the standard methods. We construct the coreset kd-tree based on $k = 30$ (number of clusters). We then would be able to generate (m, ϵ)-coreset where $m \leq k$ [19] for the in-query points.

The source code is available in http://cgl.uni-jena.de/Software/WebHome.

7.1 Databases

We use a series of randomly generated datasets for analysing the construction procedure of our data structure. However, we focused our experiments on real datasets to obtain practically relevant results. In the following, we give a brief description of all real datasets used in our evaluation.

Artificial datasets We analyze the construction of the data structures with four artificial random datasets. Instance $U1$ and $U2$ consists of $200,000$ points with 2 and 5 dimensions and $U3$ and $U4$ consists of $400,000$ and $150,000$ points with 2 and 5 dimensions are generated by taking a random point from one of 30 Gaussian distributed clusters, whose center are picked uniformly at random.

Name	# of instances	Dimension	setup time
U1	100,000	2	212.7
U2	100,000	5	1120.3
U3	400,000	2	1710.7
U4	150,000	5	2442.2
3D RN	434,874	4	5231.2
ManaN1	3,175,342	2	19842.4

Table 2: Coreset kd-tree construction

	$Query_1$ (25%)				$Query_2$ (90%)			
	Original Points		Coreset		Original Points		Coreset	
	# inst.	Time	# inst.	Time	# inst.	Time	# inst.	Time
U1	12,419	2	98	1	100,000	4	377	3
U3	49,443	2	315	1	400,000	5	531	3
Hip2	62,822	3	713	2	500,000	5	2622	3
ManaN1	376,129	8	1,549	6	3,175,342	17	3,992	13

Table 3: Coreset kd-tree range query

The Hipparcos Catalogue: An Astronomical dataset provided by observing the sky. high-precision catalogue of 117955 stars, was published in 1997 and a lower precision catalogue of more than a million stars was published at the same time. We use a set of 500,000 more accurate observation of the lower precision catalogue which is based on the stars coordination (2-dimensional) in space.

3D Road Network (3D RN) This dataset was constructed by adding elevation information to a 2D road network in North Jutland, Denmark. This dataset has 434874 instances with 4 attributes with real value.

ManaN1 Is a geographical dataset based on the polar coordination of the observations of the green area on the earth consist of $3,175,342$ points with 2 dimension.

The implementation is using C++ which is compiled by gcc. We query two ranges on the tree and then compute the coreset points and by running an standard kmeans++ algorithm on the reported coreset we record the spent time and clustering cost. The standard approach is to reach the whole point set within the range without caring about the coreset representatives and apply the same kmeans++ algorithm. The construction times and the results for the range queries are shown in Table 2 and 3. Clustering results are also given in Figures 3-6.

7.1.1 Construction

For constructing a $(k, \epsilon)-$coreset of an arbitrary set S based on sampling according to D^2, it is experienced that a sample of size $m = 200k$ provides a good result [1]. So in each of these experiments, we set size of the coreset of any subtree to $m = 200k$ (i.e. $m = 6000$). The times of construction of the coreset kd-trees are given in Table 2. Most of the construction time is spent for computing a coreset range tree spends for computing a coreset plugin for the subtrees. An this depends strongly to the size of input data. If the tree depth increase by i, we have to compute a coreset for 2^i new subtrees (for large enough trees). We should mention that we only compute a coreset for large enough subtrees. For example in tree constructed on $U1$ (over

Function Main
Input: Pointset P, Coreset_Size core_set, Region region
Output: Coreset
Tree T = CREATE_CORESET_TREE(P, Coreset_Size)
return FIND_CORESET(T.root, region, k).coreset

Algorithm 1: Coreset Construction

Function CREATE_CORESET_TREE
Input: Pointset P, Coreset_Size
Output: Tree
Create a new empty tree T
foreach p *in* P **do**
| INSERT(T, p)
end
CALCULATE_CORESETS(Coreset_Size, T.root)
return T

Algorithm 2: Create Coreset Tree

100, 000 points) we copmute a coreset for only 15 nodes. This number for $U3$ is 63 and for ManaN1 is 511. The other steps of construction need a very little time in comparison to coreset computing.

7.1.2 Dependence on input size

We perform two range queries on the constructed Coreset kd-trees. As the results are shown in Figures 3-6, for bigger datasets (and for bigger range query) the size of the reported coreset is a smaller fraction of the original points. This fraction for a small dateset like $U1$ is 45% for small query 25% but for a rather big dataset $ManaN1$ is 7.8% for query 25% and is only 3% for query 25%. After running kmeans++ on the reported points, the number of the instances again influences the performance of Kmeans++. As it mentioned before, kmeans++ is slower for the bigger input data. The results for the whole process are depicted in Figures 3-6.

7.1.3 Dependence on the number of clusters

As we construct our datasructres with a fixed value of $k = 30$, the effect of this factor on the range query can not be obtained by the results. However one can supposes that for smaller amounts of k, the size of the coreset plugins (e.g. $200k$) is smaller for the auxiliary trees. However in Figures 3-6 is shown that the performance time of kmeans++ is highly depends on the number of expected clusters.

Fig 3.1: U1 query and clustering time

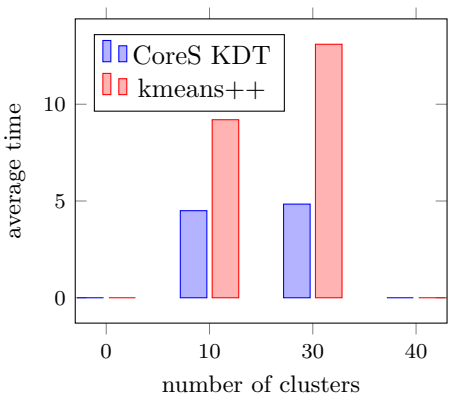
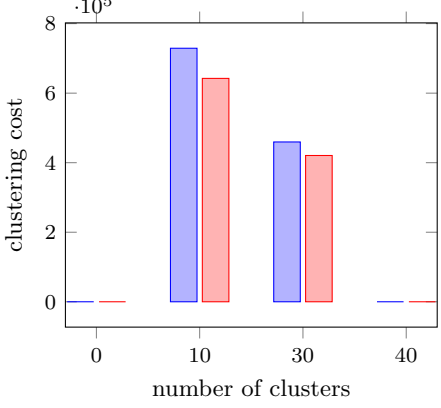


Fig 3.1: U1 clustering cost



Function INSERT
Input: Tree T, Point p
Node current = T.root
int k = 0
while *current != NULL* **do**
    if *p[k] ¡ current.point[k]* **then**
        current = current.left
    end
    else
        current = current.right
    end
    k = (k+1) mod T.dimension
end
current = new Node(p)

Algorithm 3: Insert

Function CALCULATE_CORESETS
Input: Coreset_Size, Node n
Output: Pointset
if n == *NULL* **then**
    return empty Pointset
end
Pointset left = CALCULATE_CORESETS(cs_size, n.left)
Pointset right = CALCULATE_CORESETS(cs_size, n.right)
Pointset together = left + right + n.point
n.coreset = CREATE_CORESET(together, cs_size)
return together

Algorithm 4: Calculate Coreset

Fig 4.1: U3 query and clustering time

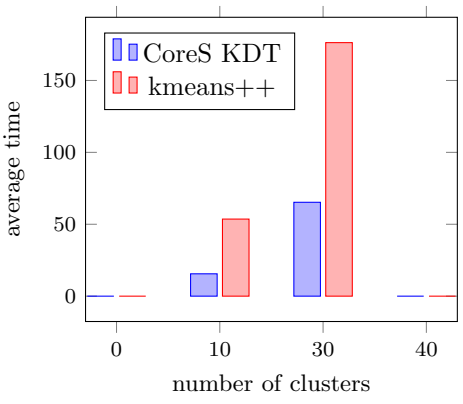
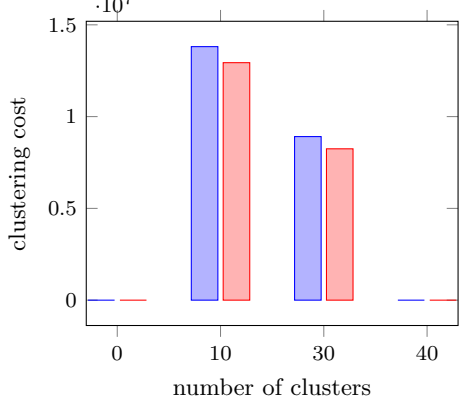


Fig 4.2: U3 clustering cost



```
Function FIND_CORESET
Input: Node current, Region region, int k
Output: Coreset, boolean
if current == NULL then
  return ({}, true)
end
else
  if current.point[k] ¡ region.min[k] then
    coreset = FIND_CORESET(current.right, region, (k+1) mod
    T.dimension).coreset
    return (coreset, false)
  end
  if current.point[k] ¿= region.max[k] then
    coreset = FIND_CORESET(current.left, region, (k+1) mod
    T.dimension).coreset
    return (coreset, false)
  end
  else
    left = FIND_CORESET(current.left, region, (k+1) mod T.dimension)
    right = FIND_CORESET(current.right, region, (k+1) mod
    T.dimension)
    if left.boolean AND right.boolean AND (current.point is inside region)
    then
      return (current.coreset, true)
    end
    else
      coreset = left + right
      if current.point is inside region then
        coreset = coreset + current.point-0u
      end
      return (coreset, false)
    end
  end
end
```

Algorithm 5: Find Coreset

Fig 5.1: Hip2 query and clustering time

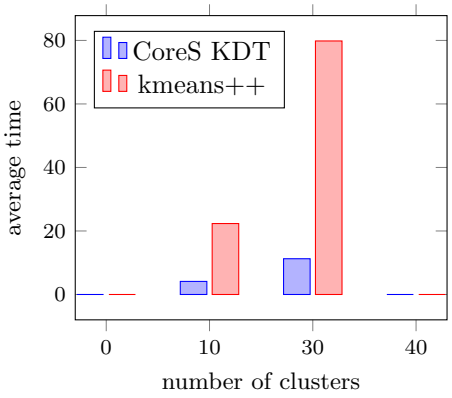
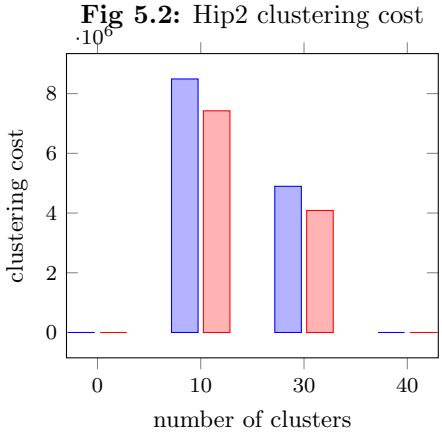


Fig 5.2: Hip2 clustering cost



Function CREATE_CORESET
the boolean denotes if the subtree of the current node contains only points within the region **Input**: Pointset P, Coreset_Size
Output: Pointset
see Algorithm 4.1 and 4.2 in "StreamKM++: A Clustering Algorithm for Data Streams"

Algorithm 6: Create Coreset

Fig 6.1: ManaN1 query and clustering time

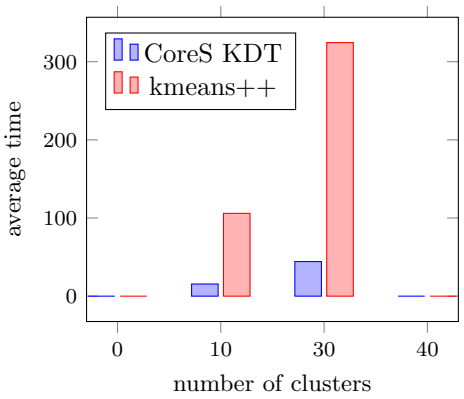
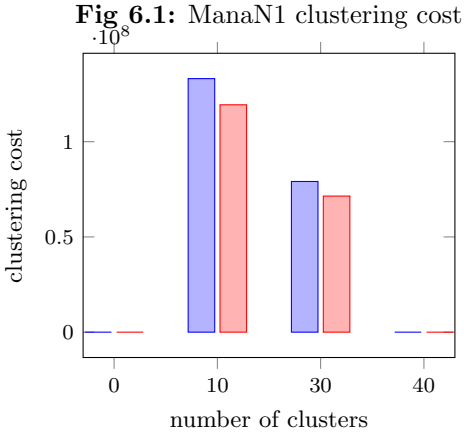


Fig 6.1: ManaN1 clustering cost



References

[1] Marcel R. Ackermann, Marcus Märtens, Christoph Raupach, Kamil Swierkot, Christiane Lammersen, and Christian Sohler. Streamkm++: A clustering algorithm for data streams. *J. Exp. Algorithmics*, 17(1):2.4:2.1–2.4:2.30, May 2012.

[2] Pankaj K. Agarwal and Jeff Erickson. Geometric range searching and its relatives, 1999.

[3] Lars Arge, Mark De Berg, Herman Haverkort, and Ke Yi. The priority r-tree: A practically efficient and worst-case optimal r-tree. *ACM Trans. Algorithms*, 4(1):9:1–9:30, March 2008.

[4] David Arthur and Sergei Vassilvitskii. k-means++: the advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '07, pages 1027–1035, Philadelphia, PA, USA, 2007. Society for Industrial and Applied Mathematics.

[5] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The r*-tree: an efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, SIGMOD '90, pages 322–331, New York, NY, USA, 1990. ACM.

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

[7] Jon Louis Bentley and Jerome H. Friedman. Data structures for range searching. *ACM Comput. Surv.*, 11(4):397–409, December 1979.

[8] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag TELOS, Santa Clara, CA, USA, 3rd ed. edition, 2008.

[9] Mihai Bdoiu, Sariel Har-peled, and Piotr Indyk. Approximate clustering via core-sets. In *In Proc. 34th Annu. ACM Sympos. Theory Comput*, pages 250–257, 2002.

[10] Philippe Chanzy, Luc Devroye, and Carlos Zamora-cura. Analysis of range search for random k-d trees. *Acta Informatica*, 37:355–383, 1999.

[11] Ke Chen. On coresets for k-median and k-means clustering in metric and euclidean spaces and their applications. *SIAM J. Comput.*, 39(3):923–947, August 2009.

[12] Douglas Comer. The ubiquitous b-tree. *ACM Computing Surveys*, 11:121–137, 1979.

[13] Mount David. Lecture notes in computer graphics, 2012.

[14] Luc Devroye, Jean Jabbour, and Carlos Zamora-cura. Squarish k-d trees. *SIAM Journal on Computing*, 30:1678–1700, 2000.

[15] Schmidt Melanie Feldman, Dan and Christian Sohler. Turning big data into tiny data: Constant-size coresets for k-means, pca and projective clustering. In *Proceedings of the 24th annual ACM-SIAM symposium on Discrete algorithms*, SODA '13, Philadelphia, PA, USA, 2013. Society for Industrial and Applied Mathematics.

[16] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3(3):209–226, September 1977.

[17] Antonin Guttman. R-trees: a dynamic index structure for spatial searching. *SIGMOD Rec.*, 14(2):47–57, June 1984.

[18] S. Har-Peled and A. Kushal. Smaller coresets for k-median and k-means clustering. pages 126–134, 2005. 04/small_coreset/.

[19] Sariel Har-Peled and Soham Mazumdar. On coresets for k-means and k-median clustering. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, STOC '04, pages 291–300, New York, NY, USA, 2004. ACM.

[20] Michal Kratky, Vaclav Snasel, Jaroslav Pokorny, and Pavel Zezula. Efficient processing of narrow range queries in multi-dimensional data structures. In *Database Engineering and Applications Symposium, 2006. IDEAS '06. 10th International*, pages 69 –79, dec. 2006.

[21] S. Lloyd. Least squares quantization in pcm. *IEEE Trans. Inf. Theor.*, 28(2):129–137, September 2006.

[22] John T. Robinson. The k-d-b-tree: a search structure for large multidimensional dynamic indexes. In *Proceedings of the 1981 ACM SIGMOD international conference on Management of data*, SIGMOD '81, pages 10–18, New York, NY, USA, 1981. ACM.

[23] Nick Roussopoulos and Daniel Leifker. Direct spatial search on pictorial databases using packed r-trees. In *Proceedings of the 1985 ACM SIGMOD international conference on Management of data*, SIGMOD '85, pages 17–31, New York, NY, USA, 1985. ACM.

[24] Timos K. Sellis. Efficiently supporting procedures in relational database systems. *SIGMOD Rec.*, 16(3):278–291, December 1987.

[25] Qingxiu Shi and Bradford G. Nickerson. k-d range search with binary patricia tries. Technical report, 2004.

[26] Tomas Skopal and et al. On range queries in universal b-trees, 2003.