CGL
Computational Geometric Learning

PROBI: A Heuristic for the probabilistic k-median problem

Hendrik Fichtenberger Melanie Schmidt

CGL Technical Report No.: 69

Part of deliverable:	WP-1/RTD
Site:	TUDO
Month:	36

Research partially supported by the European Commission within the 7th Framework Programme (2010–2013) under contract nr. IST-255827 and by DFG grant SO 514/4-3.

Abstract. We develop the heuristic PROBI for the probabilistic Euclidean k-median problem based on a coreset construction by Lammersen et al. [29]. Our algorithm computes a summary of the data and then uses an adapted version of k-means++ [5] to compute a good solution on the summary. The summary is maintained in a data stream, so PROBI can be used in a data stream setting on very large data sets.

We experimentally evaluate the quality of the summary and of the computed solution and compare the running time to state of the art data stream clustering algorithms.

1 Introduction

Clustering is a basic machine learning task: Partition a set of objects into subsets of similar objects. Geometric clustering is used to cluster sets of points, and the (dis)similarity between the points is then measured by a distance function. Geometric clustering problems then differ by the choice of the distance function. A very natural and popular distance function is the Euclidean k-median distance function that measures the dissimilarity between two points by their Euclidean distance, i. e., the input to this problem consists of points from the Euclidean space $\mathbb{R}^d$.

The Euclidean k-median problem is NP-hard [33], but it allows for an arbitrarily good approximation with a polynomial time approximation scheme (PTAS). The first PTAS was proposed by Arora et al. [4] and used the technique of the famous PTAS for Euclidean TSP by Arora [3]. Subsequently, many PTAS for the Euclidean k-median were developed, achieving better and better running times [7, 11, 13, 14, 17, 20, 21, 23, 24, 28, 30]. The algorithm by Kumar, Sabharwal and Sen [28] achieves a running time of $\mathcal{O}(2^{(k/\varepsilon)^{O(1)}}dn)$ which is linear for constant k and ε and has in particular a notably small dependence on the dimension.

However, in the presence of constantly increasing amounts of data arising from physics, social science or biology, algorithms with linear running time are not always fast enough. Experiments like the Large Hadron Collider generate so much data that even reading them more than once is time consuming. Consequently, *data stream algorithms* have risen to quite some popularity. Here, only one pass over the data is allowed (without any assumptions on the ordering of the data), and the algorithm is only allowed to store a small amount, e. g., the storage size should be polylogarithmic in the input size or even independent of the length of the stream.

Among the above cited algorithms, [11, 13, 14, 17, 20, 21, 23, 30] can also be used in a data stream. A very popular strategy to develop a streaming algorithm, used by all the cited streaming algorithms except [17] and [23], is to design an algorithm which computes a summary of the data and then to turn this algorithm into a streaming algorithm by the use of a technique called Merge & Reduce. This approach leads to an approximation algorithm if the summary is a *coreset*.

In the setting of Euclidean k-median clustering, a coreset is a small set S of weighted points from $\mathbb{R}^d$ which meets the following requirement. For every choice of k centers from $\mathbb{R}^d$, the sum of the weighted distances of all points in S to their closest center in C is a $(1+\varepsilon)$-approximation of the sum of the distances of all points in P to their nearest center in C. Such a coreset has the nice property that the union of two coresets is a coreset again. So a reasonable idea is to partition the input data into chunks, compute a coreset for each chunk and union the resulting coresets. Whenever this union grows to large, it is reduced by applying the coreset construction again. However, this might inccur an additional error, so it has to be done a little more careful. This is exactly what the Merge & Reduce technique does.

Merge & Reduce goes back to [9] and was first applied to clustering problems in [21]. By computing coresets of coresets in a tree-like fashion, Merge & Reduce makes sure that no point takes part in more than $\log n$ reduction steps, thus bounding the error that is incurred. To make up for the (bounded) additional error, Merge & Reduce requires that the coresets are by a poly-logarithmic factor larger than the coresets computed by the original coreset construction. The currently best streaming algorithm is using this approach [13] and achieves a space complexity of only $\mathcal{O}(k \cdot \log(1/\varepsilon) \cdot \log^4 n/\varepsilon^3)$.

When implementing algorithms, their theoretical guarantee can sometimes be misleading. This is in particular true for streaming settings and large data settings in general, because high constants in the running time directly make algorithms infeasible for huge amounts of data. Fast heuristics are an alternative, but give no guarantee for good solutions, so the question is whether there exist good trade-offs between practical needs and theoretical guarantees.

This trade-off has been investigated for the related k-means clustering problem. For the non streaming version, the k-means++ algorithm [5] is an improved version of the most famous heuristic for the k-means problem, Lloyd's algorithm [31], which has a reasonable theoretical guarantee but only adds a short initialization to Lloyd's algorithm and is thus still fast. In the streaming setting, StreamKM++ [1] gives a practical streaming implementation of k-means++. It can be proved that Stream-KM++ computes a coreset. On the coreset, it applies k-means++. BICO [15] also computes a coreset and uses k-means++ on the coreset, but by avoiding the Merge & Reduce technique and instead building upon a data structure by the famous streaming heuristic BIRCH [39], it achieves much better running times than Stream-KM++ while computing solutions of the same quality.

We further investigate the trade-off between worst-case guarantees and empirical performance in the setting of *probabilistic* clustering. In addition to being large, nowadays data often comes with uncertainty. For example, data collected by sensors usually has faults. But also data which is originally precise can lead to data sets with uncertainty, for example if data bases are joined and it is not certain which entities in the different data bases are the same.

The problem to cluster uncertain data has recently triggered the development of new heuristics [10,19,25,26,35,38]. In particular, [10] and [35] generalize Lloyd's algorithm to a probabilistic setting. For a survey on different heuristic approaches, see the paper by Aggarwal and Yu [2]. A theoretic study of probabilistic clustering was done by Cormode and McGregor [12]. They investigate different clustering formulations, in particular they give a $(3+\varepsilon)$-approximation algorithm for the probabilistic Euclidean k-median problem. Guha and Munagala improve one of their results on so-called k-center clustering [18].

For probabilistic clustering, only one coreset construction is known. Lammersen et al. [29] give a reduction for general metric k-median clustering to the deterministic case. Of course this would also work in the Euclidean setting because the Euclidean distance is a metric, but the reduction would only reduce it to a deterministic metric k-median problem, and those usually assume a finite set as the ground set for possible centers. Thus, [29] additionally includes an algorithm specifically for the Euclidean k-median problem, which is a generalization of the coreset construction by Chen [11].

However, their algorithm consists of many nested subroutines provided by previous results from clustering theory, and implementing it does not immediately result in an efficient or even reasonably fast algorithm. This report deals with the question how the algorithm in [29] can be heuristically modified in order to be implementable. As there is no theoretical justification for the modifications, we do not expect optimal results. However, we believe that our implementation is a good step in the development of an efficient algorithm for the probabilistic Euclidean k-median problem.

2 Preliminaries

First, we define the *weighted probabilistic Euclidean k–median–clustering problem.* Notice that Cormode and McGregor [12] refer to our scenario as the *assigned* Euclidean k-median problem. Let $\mathcal{X} := \{x_1, \dots, x_m\} \subset \mathbb{R}^d$ be a finite set of m points from the d–dimensional Euclidean space $\mathbb{R}^d$, and let $\mathcal{V} := \{v_1, \dots, v_n\}$ be a set of n *nodes*, where each node v_i follows an independent probability distribution $\mathcal{D}_i$ over $\mathcal{X}$. For any $i \in [n]$ and any $j \in [m]$, we denote the probability that the node v_i is realized at x_j by p_{ij}. We denote the total probability that v_i is realized by $p_i := \sum_{j=1}^{m} p_{ij}$. We assume that $p_i \leq 1$, which means that with probability $1 - p_i$ the node v_i is not realized.

Definition 1 (Weighted probabilistic Euclidean k–Median [12]). *Given $k \in \mathbb{N}$ and a positive weight function* $\mathrm{w} : \mathcal{V} \to \mathbb{R}_{\geq 0}$ *on the set of nodes $\mathcal{V}$, the* weighted probabilistic Euclidean k–median–clustering problem *for the set of nodes $\mathcal{V}$ is to find a set $\mathcal{C} := \{c_1, \dots, c_k\} \subset \mathbb{R}^d$ of k cluster centers and an assignment $\rho : \mathcal{V} \to \mathcal{C}$ such that the expected k–median clustering cost*

$$\mathrm{E}[\mathrm{cost}_{\mathrm{w}}(\mathcal{V}, \mathcal{C}, \rho)] := \sum_{i=1}^{n} w_i \sum_{j=1}^{m} p_{ij}\, \mathrm{d}(x_j, \rho(v_i))$$

is minimized.

For a given instance, let p_{min} be the smallest realization probability, i.e. $\min_{v_i \in \mathcal{V}, x_j \in \mathcal{X}} p_{ij}$, and let w_{min} be the minimum of the smallest weight and 1, i.e. $w_{min} := \min\{\min_{v_i \in \mathcal{V}} w(v_i), 1\}$. We denote the expected weight of all nodes by $W := \sum_{v_i \in \mathcal{V}} \mathrm{w}(v_i) p_i$.

Next, we give a definition of a *coreset* for the weighted probabilistic Euclidean k–median–clustering problem. Our definition restricts both the number of nodes in the coreset and the size of the probability distributions describing these points. Let $\mathcal{U} := \{u_1, \dots, u_s\}$ be a set of s nodes where each $u_o \in \mathcal{U}$ follows an independent probability distribution $\mathcal{D}'_o$ over $\mathcal{X}$. For any $o \in [s]$ and any $j \in [m]$, we denote the probability that u_o is realized at x_j by p'_{oj}. We denote the total probability that u_o is realized by $p'_o := \sum_{j=1}^{m} p'_{oj}$.

Definition 2 (Coreset for weighted probabilistic Euclidean k–Median [29]). *Given the set of nodes $\mathcal{V}$, let $\mathcal{U} : \{u_1, \dots, u_s\} \subseteq \mathcal{V}$ be a weighted set of nodes with positive weight function* $\mathrm{w}' : \mathcal{U} \to \mathbb{R}_{\geq 0}$, *and let $\mathcal{D}' := \{\mathcal{D}'_1, \dots, \mathcal{D}'_s\}$ be a set of s probability distributions over $\mathcal{X}$ defining the distribution of nodes in $\mathcal{U}$. Given $k \in \mathbb{N}$, a positive weight function* $\mathrm{w} : \mathcal{V} \to \mathbb{R}_{\geq 0}$ *on the set of nodes $\mathcal{V}$ and a precision parameter ϵ, $0 < \epsilon \leq 1$, the set $\mathcal{U}$ is called (k, ϵ)–coreset of $\mathcal{V}$ for the weighted probabilistic Euclidean k–median–clustering problem if, for each $\mathcal{C} \subset \mathbb{R}^d$ of size $|\mathcal{C}| = k$, we have*

$$\left| \min_{\rho:\mathcal{U}\to\mathcal{C}} \mathrm{E}_{\mathcal{D}'}[\mathrm{cost}_{w'}(\mathcal{U}, \mathcal{C}, \rho)] - \min_{\rho:\mathcal{V}\to\mathcal{C}} \mathrm{E}_{\mathcal{D}'}[\mathrm{cost}_{w}(\mathcal{V}, \mathcal{C}, \rho)] \right| \leq \epsilon \cdot \min_{\rho:\mathcal{V}\to\mathcal{C}} \mathrm{E}_{\mathcal{D}'}[\mathrm{cost}_{w}(\mathcal{V}, \mathcal{C}, \rho)].$$

Bicriteria approximations are a widely used relaxation of typical approximation algorithms, where in addition to the quality, the number of centers k does not have to be matched exactly but only approximately. We give a formal definition of a bicriteria approximation for the weighted probabilistic k–median–clustering problem.

Definition 3 (Bicriteria approximation). *Given $k \in \mathbb{N}$, a positive weight function* $\mathrm{w} : \mathcal{V} \to \mathbb{R}_{\geq 0}$ *and $\alpha, \beta \geq 1$, $\mathcal{A}$ is referred to as $[\alpha, \beta]$–approximation of the optimal center set $\mathcal{C}$, if*

$$\min_{\rho:\mathcal{V}\to\mathcal{C}} \mathrm{E}_{\mathcal{D}'}[\mathrm{cost}_{w}(\mathcal{V}, \mathcal{A}, \rho)] \leq \alpha \min_{\rho:\mathcal{V}\to\mathcal{C}} \mathrm{E}_{\mathcal{D}'}[\mathrm{cost}_{w}(\mathcal{V}, \mathcal{C}, \rho)]$$

$$|\mathcal{A}| \leq \beta k.$$

This report is organized as follows. In Section 3, we describe the coreset construction for the Euclidean probabilistic k-median problem by Lammersen, Schmidt and Sohler [29]. In Section 4, we describe how this algorithm can be modified in order to be efficiently implementable. In Section 5, we evaluate the implementation empirically.

3 Coreset for probabilistic Euclidean k–Median

In this section, we review the algorithm by Lammersen, Schmidt and Sohler [29] to compute a coreset for the probabilistic Euclidean k-median problem in data streams. First, we describe the actual coreset construction. Second, we recall how to embed the coreset construction into a data stream setting in order to compute a coreset in a stream.

3.1 Coreset construction

The coreset construction in [29] consists of five steps. We give a detailed pseudo code as Algorithm 1.

1. Construct a set $\mathcal{Y}$ containing a 2–approximation of the probabilistic 1–Median for each node $v_i \in \mathcal{V}$.
2. Construct a set $\mathcal{Y} \subseteq \mathcal{A}$ which is the center set of an $[\alpha, \beta]$–bicriteria approximation for the metric k–Median problem of $\mathcal{Y}$.
3. Partition $\mathcal{Y}$ into buckets $\mathcal{Y}_{\ell,h,a}$ that group 1–Medians (i.e. nodes) which are similar in terms of their location and their contribution to the total clustering cost.
4. Draw a sample $\mathcal{U}_{\ell,h,a} \subseteq \mathcal{Y}_{\ell,h,a}$ uniformly at random and with replacement from each partition $\mathcal{Y}_{\ell,h,a}$.
5. Approximate the probabilty distribution of each node in $\mathcal{U} := \bigcup \mathcal{U}_{\ell,h,a}$ by computing a coreset of each node $v_i \in \mathcal{U}$

Following these steps, one obtains a $(1+\epsilon)$–coreset of the input $\mathcal{V}$ with error probability δ. The following algorithms are used in [29] to implement steps 1, 2 and 5:

Step 1 The algorithm described in [28] by Kumar et al. is used to obtain a 2–approximation of the probabilistic 1–Median for a node v_i. It runs in constant time, i.e. the number of points x_j with $p_{ij} > 0$ does not affect the running time when processing node v_i. We denote this algorithm by KumarMedian::approximateOneMedian .

Step 2 Two algorithms are used. An $[\alpha, \beta]$–bicriteria approximation algorithm for the metric k–Median problem is sped up by an algorithm proposed by Indyk [22]. This algorithm is denoted by Indyk::computeCenterSet .

Step 5 The reduced probability distribution for a node v_i is computed by using an algorithm by Chen [11]. We refer to this algorithm as Chen::computeCoreset .

Fact 4 (see [29]). *Algorithm 1 can be implemented to run in*

$$\mathcal{O}\left(knm \log\left(\frac{n}{\delta}\right) \log\left(\log\left(\frac{W}{w_{min}\, p_{min}\, \epsilon}\right)\right)\right)$$

time.

Algorithmus 1 Coreset

```
function computeCoreSet($\mathcal{X}, \mathcal{V}, \mathrm{w}, k, \epsilon$)
    for all $v_i \in \mathcal{V}$ do                                   ▷ Construct $\mathcal{Y}$
        $v'_i \leftarrow \emptyset$
        for all $x_j : p_{ij} > 0$ do
            $v'_i \leftarrow v'_i \cup \{x_j \mid n \in [\,\lfloor w(v_i)p_{ij}/(w_{min}p_{min}\epsilon)\rfloor\,]\}$
        end for
        choose best one out of $\overline{r}_1$
            $y_i \leftarrow$ KumarMedian::approximateOneMedian($v'_i, 0.9$)
        $\mathrm{w}(y_i) \leftarrow \mathrm{w}(v_i)$
    end for
    $\mathcal{Y} \leftarrow \bigcup y_i$
    for all $y_i$ do                                                   ▷ Construct $\mathcal{A}$
        $y'_i \leftarrow \{y_i \mid n \in [\,\lfloor w(v_i)p_i/(w_{min}p_{min}\epsilon)\rfloor\,]\}$
    end for
    $\mathcal{Y}' \leftarrow \bigcup y'_i$
    choose best one out of $\overline{r}_2$
        $\mathcal{A} \leftarrow$ Indyk::computeCenterSet($\mathcal{Y}', k$)
    for $\ell \in \{1, \dots, \tau\}$ do                                ▷ Partition $\mathcal{Y}$
        $\mathcal{Y}_\ell \leftarrow \{y \in \mathcal{Y} \mid \arg\min_{i \in [\tau]} \mathrm{d}(y, a_i) = a_\ell\}$
        for $h \in \{0, \dots, \nu\}$ do
            $\mathcal{Y}_{\ell,h} \leftarrow \begin{cases} \mathcal{Y}_\ell \cap \mathrm{ball}(a_\ell, R) & h = 0 \\ \mathcal{Y}_\ell \cap [\mathrm{ball}(a_\ell, 2^h R) \setminus \mathrm{ball}(a_\ell, 2^{h-1}R)] & h \geq 1 \end{cases}$
                                        ▷ $ball(p, r)$ is the open ball of radius $r$ centered at $p$
            for $a \in \{0, \dots, \nu\}$ do
                $\mathcal{Y}_{\ell,h,a} \leftarrow \begin{cases} \{y_i \in \mathcal{Y}_{\ell,h} \mid \sum_{x_j \in \mathcal{X}} (p_{ij}/p_i)\,\mathrm{d}(x_j, y_i) \leq R\} & a = 0 \\ \{y_i \in \mathcal{Y}_{\ell,h} \mid 2^{a-1} < \sum_{x_j \in \mathcal{X}} (p_{ij}/p_i)\,\mathrm{d}(x_j, y_i) \leq 2^a R\} & a \geq 1 \end{cases}$
            end for
        end for
    end for
    $\mathcal{U}_{\ell,h,a} \leftarrow$ weightedSamplingWithReplacement($\mathcal{V}_{\ell,h,a}, s$)
    $\mathcal{U} \leftarrow \bigcup_{\ell,h,a} \mathcal{U}_{\ell,h,a}$
    for all $u_j \in \mathcal{U}$ do                        ▷ Approximate probability distributions
        $u'_j \leftarrow$ Unweighted multiset: see lines 3–6
        $\hat{u}_j \leftarrow$ Chen::computeCoreset($u'_j, 1, \epsilon, \delta\,/\,n$)
    end for
    $\hat{\mathcal{U}} \leftarrow \bigcup_j \hat{u}_j$
    return $\hat{\mathcal{U}}$
end function
```

3.2 Streaming Algorithm

Lammersen et al. use the Merge & Reduce technique [9,21] to enable Algorithm 2 to work in a data stream. Here, $\mathcal{V}$ is given as a stream of n weighted nodes. Each node v_i is given as a consecutive chunk in the data stream that is a sequence of up to m point–probability pairs in worst case order representing the discrete probability distribution $\mathcal{D}_i$ of the node v_i. More precisely, the nodes are organized in a small number of coresets, each representing $2^\ell N$ nodes (for some integer ℓ and a fixed constant N). Every time when two coresets representing the same number of nodes exist, we take the union (merge) and create a new coreset (reduce). The construction is based on the following fact:

Fact 5. *1. If $\mathcal{U}_1$ and $\mathcal{U}_2$ are (k,ϵ)–coresets for disjoint sets $\mathcal{V}_1$ and $\mathcal{V}_2$, then $\mathcal{U}_1 \cup \mathcal{U}_2$ is a (k,ϵ)–coreset for $\mathcal{V}_1 \cup \mathcal{V}_2$.*

2. If $\mathcal{U}_1$ is a (k,ϵ_1)–coreset for $\mathcal{U}_2$ and $\mathcal{U}_2$ is a $(k,\epsilon_2$–coreset for $\mathcal{U}_3$, then $\mathcal{U}_1$ is a $(k,(1+\epsilon_1)(1+\epsilon_2)-1)$–coreset for $\mathcal{U}_3$.

The idea is as follows. We maintain buckets $B_0, B_1, \ldots$ which are created on demand. Bucket B_0 can store between 0 and N nodes. For $\ell \geq 1$, bucket B_ℓ is either empty or stores a coreset U_ℓ of approximately N coreset nodes representing $2^{\ell-1}$ nodes from the data stream. Next, we explain the method in detail.

All nodes in the data stream are processed in the same way. Let v_i be the i–th node read from the data stream. Then, v_i is inserted into bucket B_0. If bucket B_0 is full, then all nodes from B_0 are moved to bucket B_1. If bucket B_1 is empty, we are done. Otherwise, we compute a coreset $\mathcal{U}_2$ from the union of the approximately $2N$ nodes stored in B_0 and B_1 using algorithm 2. Afterwards, both buckets B_0 and B_1 are emptied and the approximately N coreset nodes from $\mathcal{U}_2$ are moved into bucket B_2. If B_2 is empty, we are done. Otherwise, we compute a coreset U_3 from the union of the approximately $2N$ coreset nodes stored in B_1 and B_2 using algorithm 2. Then, buckets B_1 and B_2 are emptied and the approximately N coreset nodes from $\mathcal{U}_3$ are moved into bucket B_3. If bucket B_3 is empty, we are done. Otherwise, we repeat this process until we reach an empty bucket. The final coreset is constructed by reducing B_0 and returning the union of all buckets.

4 Implementation and Runtime Improvements

Now, we describe how we modify Algorithm 1 with the goal of a practically efficient algorithm. The implementation will be available at http://cgl.uni-jena.de/Software/WebHome.

As this is the first data stream algorithm for this problem, we do not expect a perfectly behaving algorithm. However, we would like to achieve that the algorithm is efficient at least for inputs with sparse or only moderatly dense nodes. This in particular means that we concentrate on the core algorithm and ignore step 5 for now.

Algorithm 1 contains several subproblems that have to be solved, and [29] points to asymptotically efficient algorithms for all of them. However, these algorithms are not necessarily efficient in practice. For example, the approximation algorithm to compute the 1-median has an asymptotic running time of $\mathcal{O}(1)$, but the constant is quite large. Especially for sparse nodes, introducing a very large constant for every 1-median computation heavily restricts the number of nodes that can be processed in a given amount of time. The situation is similar for the bicriteria approximation algorithm.

Notice that because the coreset construction is used within a Merge & Reduce framework, we know an upper bound on the input size for each subproblem. We will restrict the size of the chunks even more and use subroutines that work well on inputs of this known size.

In the following, we describe heuristic approaches to replace the steps of Algorithm 1 in order to enable it to work for inputs of moderate up to huge size.

4.1 Construction of $\mathcal{Y}$

Finding the 1-median or *geometric median* of a point set is also known as the *Fermat-Weber problem* and has a long history. In fact, it is impossible to construct the 1-median using only a ruler and a compass (*straight-edge and compass constructions*) [34], and the problem is not solvable by *radicals* [8], i. e., it is not expressable in terms of $(+, -, *, /, \sqrt[k]{\cdot})$ over $\mathbb{Q}$.

One popular approach to approximately determine the 1-median is an iterative approach known as *Weiszfeld's algorithm* [36, 37]. It defines a sequence $y^{(k)}$ which converges to the 1–Median of the point set $\mathcal{X}$ [27] and is defined by

$$y^{(k+1)} = \left(\sum_{x_j \in \mathcal{X}} \frac{1}{\|x_j - y^{(k)}\|} x_j \right) \Bigg/ \left(\sum_{x_j \in \mathcal{X}} \frac{1}{\|x_j - y^{(k)}\|} \right).$$

In the context of our problem to compute the set $\mathcal{Y} = \{y_1, \ldots, y_n\}$ of 1–Medians of all nodes v_i, we define the recurrence $y_i^{(\nu)}$ for every node v_i by

$$y_i^{(\nu+1)} = \left(\sum_{x_j \in \mathcal{X}} \frac{w_{ij}}{\|x_j - y_i^{(\nu)}\|} x_j \right) \Bigg/ \left(\sum_{x_j \in \mathcal{X}} \frac{w_{ij}}{\|x_j - y_i^{(\nu)}\|} \right). \tag{1}$$

For each node v_i, its 1–Median y_i is approximated as follows: We choose the center of gravity $(\sum_{x_j \in \mathcal{X}} w_{ij} x_j) / (\sum_{x_j \in \mathcal{X}} w_{ij})$ as initial point y_i^0. As long as

$$\|y_i^{(\nu)} - y_i^{(\nu-1)}\| \,/\, \|y_i^{(\nu-1)} - y_i^{(\nu-2)}\| \leq 0.1 \text{ and } k < 15 \tag{2}$$

is satisfied, the successor $y_i^{\nu+1}$ of y_i^{ν} is computed by evaluating (1) and k is incremented by one. Finally, when (2) is violated, we conclude by returning y_i^{ν}. The number of iteration k is chosen experimentally. In fact, on the present data the process usually computes good solutions after very few iterations.

There is one special case we have to take care of: The iteration may arrive at a point $y_i^{(\nu)} = x_j \in \mathcal{X}$, thus no successor is defined (because $\|x_j - y_i^{(\nu)}\|$ equals zero). Since this happens very rarely, we abort the iteration and use Kumar's algorithm as fallback (see section 4.1) in this case. Notice that for non-sparse nodes, using Kumar's algorithm is a good idea in any case because of its (high but) constant running time.

We will refer to this algorithm as Weiszfeld::approximateOneMedian.

4.2 Construction of $\mathcal{A}$

Lloyd's algorithm [31] for the k-means problem is probably the most used clustering algorithm. Algorithms of similar structure are also used for other clustering functions, and we adapt it to the probabilistic Euclidean k-median problem.

The original algorithm starts with an initial set of k centers from $\mathbb{R}^d$ and then alternately performs two steps until a good enough solution is found. Step 1 assignes every point to its closest center in the current center set S. Step 2 computes the center of gravity for all subsets and replaces S by the set of these k centers. Notice that both steps can only improve the cost, because assigning each point to its closest center can only be cheaper than assigning it to any other center, and for given subsets of points with the same center, the center of gravity is the optimal choice as a center. However, there are inputs where the k-means algorithm needs an exponential number of iterations, and it is also known that it may converge to a local optimum.

The k-means++ [5] algorithm improves this behaviour by adding a procedure to compute a good initial set of centers. It chooses an initial set iteratively, starting with one point chosen uniformly at random. In every step, the distance of each point to the so-far chosen points is computed. Then, a point is chosen randomly and added to the center set, and the probability of each point to be chosen is proportional to the squared distance of the point.

To adapt this algorithmic idea for the (deterministic) Euclidean k-median problem, we need a subroutine to compute a 1-median of a subset of points. As in Section 4.1, we use Weiszfeld's algorithm to find an estimate for the 1-median. This might not result in the best 1-median, but should on average produce a 1-median that is better than the previous center of the subset.

The adaptation of the k-means++ seeding procedure is straightforward, we choose k centers iteratively and base the probability distribution on the Euclidean k-median cost instead of the squared Euclidean distance. We get the following algorithm, which we call LloydMedian::computeCenterSet:

1. Draw the first center c_1 uniformly at random from $\mathcal{Y}$
2. Draw a new center c_i by choosing $y \in \mathcal{Y}$ with probability
$$\frac{\min_{c \in \{c_1^{(0)}, \dots, c_{i-1}^{(0)}\}} \mathrm{d}(y, c)}{\sum_{\tilde{y} \in \mathcal{Y}} \min_{c \in \{c_1^{(0)}, \dots, c_{i-1}^{(0)}\}} \mathrm{d}(\tilde{y}, c)}$$
3. Repeat step 2, until we have drawn k centers $\mathcal{A}^{(0)} = \{c_1, \dots, c_k\}$
4. Assign each point $y_i \in \mathcal{Y}$ to the neareast center in $\mathcal{A}^{(\nu)}$
5. Compute an estimate $c_i^{(\nu)}$ of the 1–Median of each cluster by applying (1) as described above and set $\mathcal{A}^{(\nu+1)} := \{c_1^{(\nu)}, \dots, c_k^{(\nu)}\}$
6. Repeat steps 4 and 5, until no point is assigned to a new center or 10 iterations are completed

4.3 Construction of the final coreset

After constructing $\mathcal{A}$, we partition $\mathcal{Y}$ into $\mathcal{Y}_{\ell,h,a}$ as in algorithm 1. From each partition $\mathcal{Y}_{\ell,h,a}$, we sample s points $\mathcal{U}_{\ell,h,a}$ with replacement, where

$$s := \max\left(\frac{200 \cdot k}{|\{y \in \mathcal{Y}_{\ell,h,a} \,:\, |y| > 0\}|},\ 1\right).$$

As mentioned before, we drop Step 5 of the algorithm by [29] and do not compute coresets for the nodes in $\mathcal{U}_{\ell,h,a}$. The union of all $\mathcal{U}_{\ell,h,a}$ constitutes the final coreset $\mathcal{U}$ of $\mathcal{V}$. Algorithm 2 outlines the changes in the original steps of algorithm 1.

Algorithmus 2 PROBI

```
    function computeCoreSet($\mathcal{X}, \mathcal{V}, \mathrm{w}, k, \epsilon$)
        for all $v_i \in \mathcal{V}$ do                                      ▷ Construct $\mathcal{Y}$
            $y_i \leftarrow$ Weiszfeld::approximateOneMedian($v_i$)
            if iteration failed then
5:              $v_i' \leftarrow \emptyset$
                for all $x_j : p_{ij} > 0$ do
                    $v_i' \leftarrow v_i' \cup \{x_j \mid n \in [\,\lfloor w(v_i)p_{ij}/(w_{min}p_{min}\epsilon)\rfloor\,]\}$
                end for
                $y_i \leftarrow$ KumarMedian::approximateOneMedian($v_i'$, 0.9)
10:         end if
            $\mathrm{w}(y_i) \leftarrow \mathrm{w}(v_i)$
        end for
        $\mathcal{Y} \leftarrow \bigcup y_i$
        $\mathcal{A} \leftarrow$ LloydMedian::computeCenterSet($\mathcal{Y}, k$)      ▷ Construct $\mathcal{A}$
15:     See algorithm 1 for partitioning of $\mathcal{Y}$
        $\mathcal{U}_{\ell,h,a} \leftarrow$ weightedSamplingWithRep($\mathcal{V}_{\ell,h,a}, s$)
        $\mathcal{U} \leftarrow \bigcup_{\ell,h,a} \mathcal{U}_{\ell,h,a}$
        return $\mathcal{U}$
    end function
```

4.4 A clustering algorithm for the probabilistic k–Median problem

In principle, every algorithm for the Euclidean probabilistic k-median problem can be used to compute a solution on the coreset computed by PROBI. However, there is no standard algorithm for solving the probabilistic Euclidean k-median problem so far. We use similar modifications of Lloyd's algorithm and k-means++ as described in Section 4.2 for the case of deterministic k-median clustering. This yields an algorithm we name P-LLOYD++ which we use for computing the actual solution.

Since centers are points, not nodes, we choose the first center randomly from $\mathcal{X}$ (to be exact, from $\{x_j \in \mathcal{X} \mid \exists\, w_{ij} > 0\}$). All remaining points x_j are chosen as centers according to their *total realization score* $\sum_{v_i \in \mathcal{V}} w_{ij}$ and their distance to the nearest of the previously chosen centers.

In the probabilistic k–Median problem we consider in this paper, nodes are assigned to a center as a whole, i.e. independent from their actual location. When partitioning the nodes, we assign each node v_i to its expected nearest center. The new center of a partition is constructed by computing the 1–Median of the realizations of all nodes in this partition. We obtain the following algorithm which we will call P-LLOYD++ in the remainder of the report.

1. Draw the first center c_1 uniformly at random from $\mathcal{X}$

2. Draw a new center c_i by choosing $x_j \in \mathcal{X}$ with probability

$$\frac{\sum_{v_i \in \mathcal{V}} \min_{c \in \{c_1^{(0)}, \ldots, c_{i-1}^{(0)}\}} w_{ij}\ \mathrm{d}(x_j, c)}{\sum_{\tilde{x_\ell} \in \mathcal{X}} \sum_{v_i \in \mathcal{V}} \min_{c \in \{c_1^{(0)}, \ldots, c_{i-1}^{(0)}\}} w_{i\ell}\ \mathrm{d}(\tilde{x_\ell}, c)}$$

3. Repeat step 2, until we have drawn k centers $\mathcal{C}^{(0)} = \{c_1, \ldots, c_k\}$

4. Assign node $v_i \in \mathcal{V}$ to the expected neareast center in $\mathcal{C}^{(\nu)}$

	BigCross	CalTech128	Census	CoverType	Tower
Number of Points	11620300	3168383	2458285	581012	4915200
Dimension	57	128	68	55	3
Number of Nodes	1162030	316838	245828	58101	491520

Table 1: Sizes and dimenions of all data sets used for the experiments.

5. Compute an estimate $c_i^{(\nu)}$ of the 1–Median of each cluster by applying (1) to all realizations in c_i^{ν} and set $\mathcal{C}^{(\nu+1)} := \{c_1^{(\nu)}, \ldots, c_k^{(\nu)}\}$
6. Repeat steps 4 and 5, until no node is assigned to a new center or 10 iterations are completed

5 Experiments

5.1 Settings and Data

Setting. All computations were performed on seven machines with the same hardware configuration (2.8 Ghz Intel E7400 with 3 MB L2 Cache and 8 GB main memory). PROBI was implemented in C++ and compiled with gcc 4.7.3.

Datasets. For the experiments, we need data sets that are large enough to test the ability of PROBI to process large amounts of data. Furthermore, as there are no data stream implementations of algorithms for the Euclidean k-median that we can compare PROBI with, we wanted to at least be able to compare the speed with implementations for similar deterministic algorithms. So we decided to create probabilistic data from the data sets used in the evaluation of BICO, which is an algorithm for the deterministic k-means problem. Of course the problems are different (which is particularly striking when comparing the situation for 1-median with the existence of an explicit formula for 1-means), but in this way we at least have an indicator how long or short the time is that PROBI needs to process the data.

All points in a data set form the set $\mathcal{X} = \{x_1, \ldots, x_m\}$. We scan the data set once to form the set of nodes $\mathcal{V} = \{v_1, \ldots, v_n\}$. Every time a chunk of 10 points $\{x_j, \ldots, x_{j+9}\}$ was read, node v_i is created with a uniform distribution over its 10 realization posititions. Overall, $\lfloor m/10 \rfloor$ nodes are constructed. Up to 9 points at the end of the data file are omitted.

We use five data sets. *Tower*, *Covertype* and *Census* are data sets provided by the UCI Machine Learning Repository [6]. *BigCross* is a subset of the Cartesian product of *Tower* and *Covertype*, created by the authors of [1] to have a very large data set. Additionally, we used a data set we call CalTech128 which is also large and has higher dimension. It consists of 128 SIFT descriptors [32] computed on the Caltech101 object database. Table 1 gives the data set sizes and dimensions as well as the resulting number of nodes $|\mathcal{V}|$.

Experiments. We ran PROBI on all five data sets with different values for k which we chose similar to the values studied for BICO. However, as our point set is smaller (because every point consists of ten possible realization positions), we did not test the values of k where the size of the summary would have been too large compared to the set of the original point set.

PROBI computes a solution for the Euclidean probabilistic k-median problem by first computing a summary and then using the adapted Lloyd algorithm. In principle, we can compare the quality of this solution with the quality of an optimal solution. However, due to the lack of an approximation algorithm with guaranteed quality, we have no optimal or near optimal solution to compare with. Instead, we used P-LLOYD++ as described in Section 4.4, ran it on the whole input set to compute a solution and compared the solution quality of PROBI with the quality of this solution on Census and CoverType.

P-LLOYD++ (at least in its intuitive implementation) is too memory consuming to compute solutions for the three larger data sets, so the tests on Census and CoverType are the main source of information when evaluating the quality of the solutions computed of PROBI. The larger data sets are still valuable to judge the speed of the algorithm. As an indication on the quality of the solutions computed for these data sets, we compared the cost of the solution on the coreset with the cost of the solution on the whole data set. This does not necessarily tell whether the solution is good (because a better solution could have a higher cost on the coreset and is thus ignored) but at least it states how accurate the summary predicts the cost for this solution.

The experiments were repeated 100 times. Tables 2 contains the mean values, Table 3 contains the median values and Table 4 contains all variance coefficients (all three in the appendix).

Quality. Due to the heuristic changes, we do not expect that PROBI computes an approximation in the sense that an optimal solution on our weighted summary is always within a $(1+\varepsilon)$-fraction of the optimal solution of the point set. However, we do expect that the cost is within a constant factor, and the experiments support this idea.

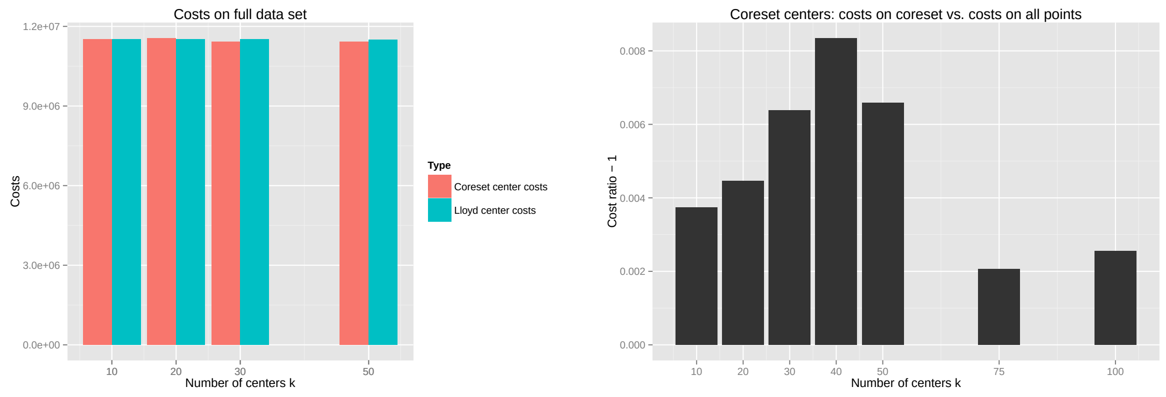

Figure 1: Census Costs. Left: Cost of PROBI solution compared with P-LLOYD++ solution (evaluated on the full set). Right: Probi solution evaluated on summary and full data set, depicted is the difference divided by the cost on the full set.

For Census, the results are excellent. For all tested k, the cost of the solution computed by PROBI is within one percent of the cost computed by P-LLOYD++ on the full data set (see the left diagram in Figure 1 and Tables 2 and 3). For CoverType, the costs of the PROBI solutions are within two times of the cost, a little better for some k (see the left diagram in Figure 2).

It is interesting to notice that the comparison of the cost of the solution on the summary and the whole point set shows a similar behaviour (see the right diagrams in Figure 1 and Figure 2). On Census, the difference is below one percent of the cost, on CoverType it is less than the cost (so the factor between the two is at most two). In particular, the difference is largest for $k = 20$, which is indeed the case where the solution quality is worst, and the quantities also look connected. This indicates that looking at how much the cost of the PROBI solution changes

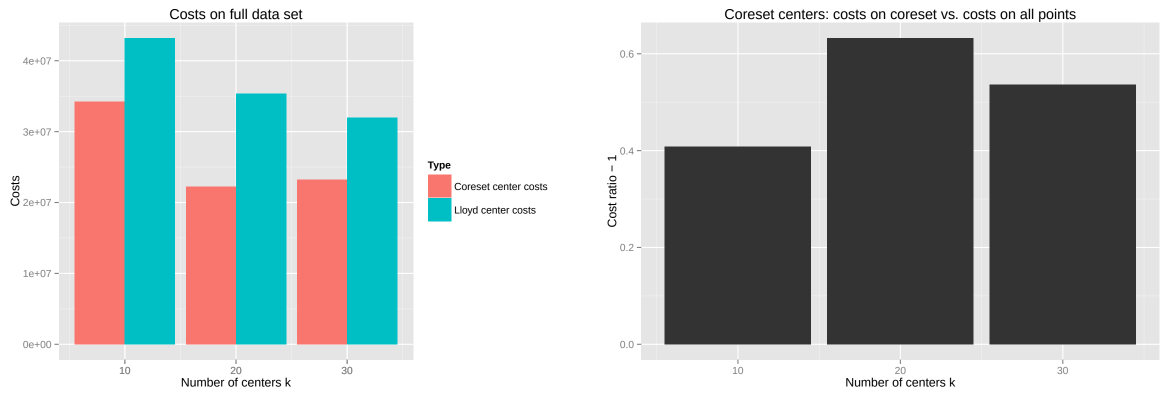

Figure 2: CoverType Costs. Diagramm explanation see Figure 1.

when evaluating it on the whole point set instead of the summary gives a hint on the solution quality.

Based on this, the results for BigCross (Figure 4), Tower (left diagramm in Figure 3) and Caltech (right diagramm in Figure 3) indicate that the solution quality might be within a factor of two as well (because the difference in the cost divided by the cost is always less than 1).

Running time. Figures 5, 6 and 7 show the running times of PROBI for all tested data sets. The time increases linearly with the number of centers and we suspect that this is due to the computation of a bicriteria approximation using P-LLOYD++, which has a relatively high running time.

Again, there are no obvious algorithms to compare the running time of PROBI with. PROBI is much faster than the adapted k-means++ algorithm P-LLOYD++, which is not surprising in light of similar results for StreamKM++ compared with k-means++ in [1]. The experiments for PROBI were performed on the same machines as those for BICO in [15]. This allows us to compare the running times. Comparing with Figures 2, 3 and 4 in [15], we see that the worst case of the ratio between PROBIs and BICOs running time is around 2 considering all cases where both algorithms were tested. This makes PROBIs running time comparable to BICO

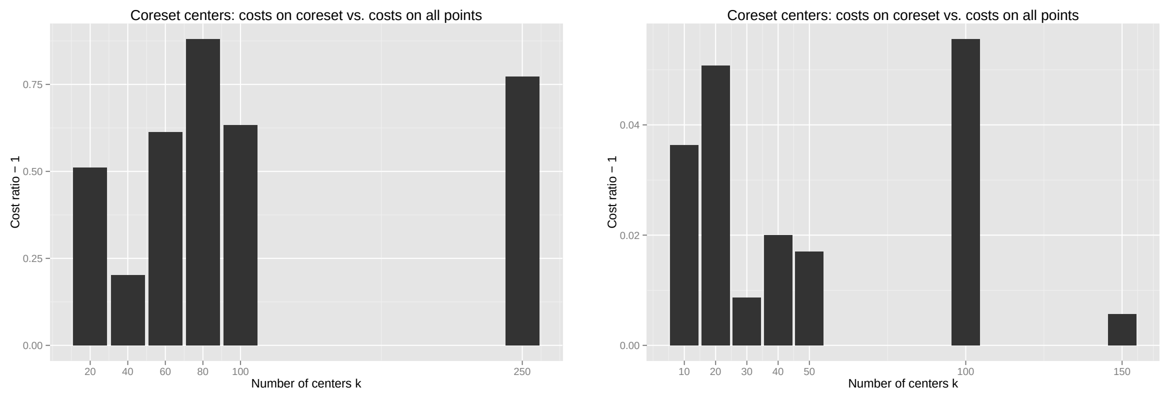

Figure 3: Tower and CalTech Costs. Diagramm explanations see Figure 4.

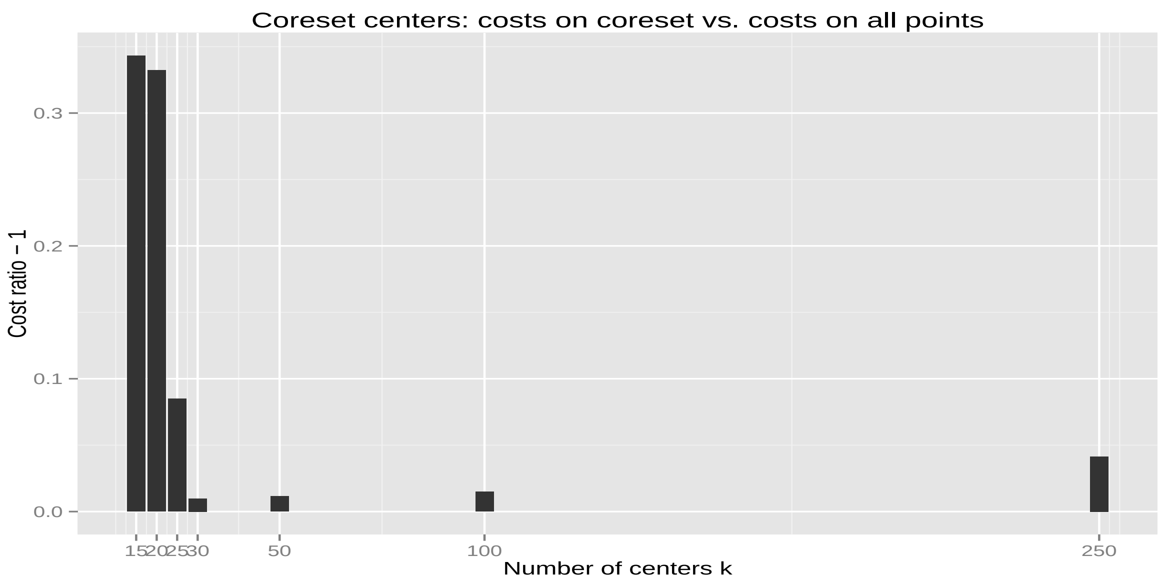

Figure 4: BigCross Costs. Diagramm depicts difference between evaluation on summary and full set, normalized by dividing by the cost on the full set.

and much faster than for example StreamKM++. Notice however that PROBIs running time increases faster than BICOs with increasing k (but linearly).

Note. This technical report is also available on ArXiv [16].

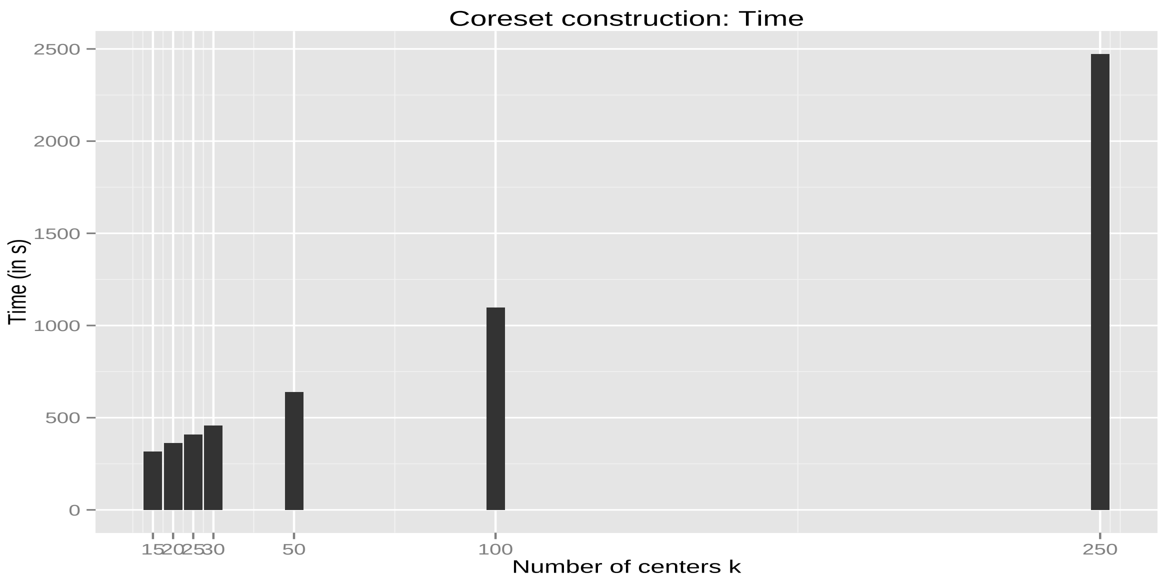

Figure 5: BigCross Running Times.

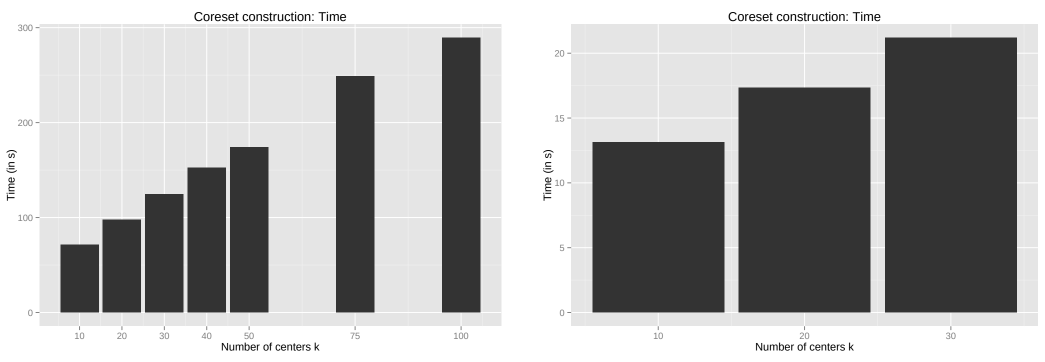

Figure 6: Census and CoverType Running Times.

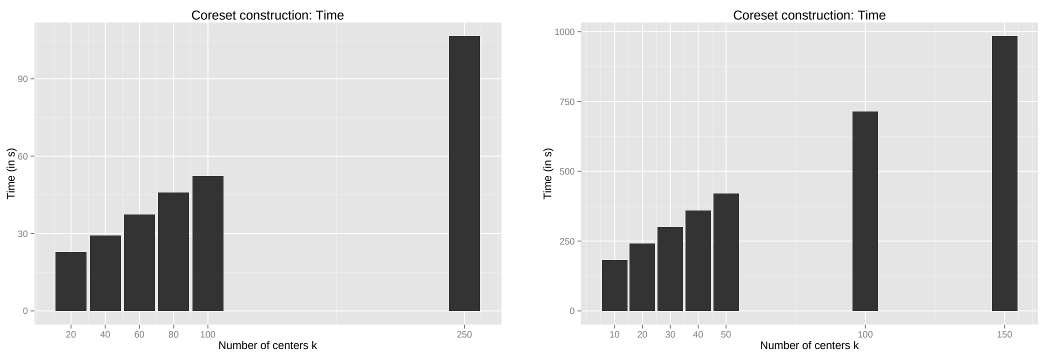

Figure 7: Tower and CalTech Running Times.

References

[1] M. R. Ackermann, M. Märtens, C. Raupach, K. Swierkot, C. Lammersen, and C. Sohler. StreamKM++: A clustering algorithm for data streams. *ACM Journal of Experimental Algorithmics*, 1(17):Article 2.4, 2012.

[2] C. C. Aggarwal and P. S. Yu. A survey of uncertain data algorithms and applications. *IEEE Transactions on Knowledge and Data Engineering*, 21(5):609 – 623, 2009.

[3] S. Arora. Polynomial time approximation schemes for euclidean traveling salesman and other geometric problems. *Journal of the ACM*, 45(5):753 – 782, 1998.

[4] S. Arora, P. Raghavan, and S. Rao. Approximation schemes for euclidean k-medians and related problems. In *Proceedings of the 30th Annual ACM symposium on Theory of Computing (STOC)*, pages 106 – 113, 1998.

[5] D. Arthur and S. 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. Society for Industrial and Applied Mathematics, 2007.

[6] K. Bache and M. Lichman. Uci machine learning repository, 2013.

[7] M. Bădoiu, S. Har-Peled, and P. Indyk. Approximate clustering via core-sets. In *Proceedings of the 34th Annual ACM symposium on Theory of Computing (STOC)*, pages 250–257, 2002.

[8] C. L. Bajaj. The algebraic degree of geometric optimization problems. *Discrete & Computational Geometry*, 3:177 – 191, 1988.

[9] J. L. Bentley and J. B. Saxe. Decomposable searching problems I. Static-to-dynamic transformation. *Journal of Algorithms*, 1(4):301–358, 1980.

[10] M. Chau, R. Cheng, B. Kao, and J. Ng. Uncertain data mining: An example in clustering location data. In *Proceedings of the 10th PAKDD*, pages 199 – 204, 2006.

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

[12] G. Cormode and A. McGregor. Approximation algorithms for clustering uncertain data. In *Proceedings of the 27th ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 191–200. 2008.

[13] D. Feldman and M. Langberg. A unified framework for approximating and clustering data. In *Proceedings of the 43rd Annual ACM Symposium on Theory of Computing (STOC)*, pages 569–578. 2011.

[14] D. Feldman, M. Monemizadeh, and C. Sohler. A PTAS for k-means clustering based on weak coresets. In *Proceedings of the 23rd Symposium on Computational Geometry (SoCG)*, pages 11 – 18, 2007.

[15] H. Fichtenberger, M. Gillé, M. Schmidt, C. Schwiegelshohn, and C. Sohler. BICO: BIRCH meets coresets for k-means clustering. In *Proceedings of the 21st European Symposium on Algorithms (ESA)*, 2013. To appear.

[16] H. Fichtenberger and M. Schmidt. Probi: A heuristic for the probabilistic k-median problem. *CoRR*, abs/1309.5781, 2013.

[17] G. Frahling and C. Sohler. Coresets in dynamic geometric data streams. In *Proceedings of the 37th Annual ACM Symposium on Theory of Computing (STOC)*, pages 209 – 217, 2005.

[18] S. Guha and K. Munagala. Exceeding expectations and clustering uncertain data. In *Proceedings of the 28th PODS*, pages 269 – 278, 2009.

[19] S. Günnemann, H. Kremer, and T. Seidl. Subspace clustering for uncertain data. In *Proceedings of the SIAM International Conference on Data Mining*, pages 385–396, 2010.

[20] S. Har-Peled and A. Kushal. Smaller coresets for k-median and k-means clustering. *Discrete & Computational Geometry*, 37(1):3–19, 2007.

[21] S. Har-Peled and S. Mazumdar. On coresets for k-means and k-median clustering. In *Proceedings of the 36th Annual ACM Symposium on Theory of Computing (STOC)*, pages 291–300. 2004.

[22] P. Indyk. Sublinear time algorithms for metric space problems. In *Proceedings of the 31st annual ACM symposium on Theory of computing*, pages 428–434. ACM, Atlanta and Georgia and United States, 1999.

[23] P. Indyk. Algorithms for dynamic geometric problems over data streams. In *Proceedings of the 36th Annual ACM symposium on Theory of Computing (STOC)*, pages 373 – 380, 2004.

[24] S. G. Kolliopoulos and S. Rao. A nearly linear-time approximation scheme for the euclidean k-median problem. *SIAM Journal on Computing*, 37(3):757 – 782, 2007.

[25] H.-P. Kriegel and M. Pfeifle. Density-based clustering of uncertain data. In *Proceedings of the 11th ACM SIGKDD*, pages 672–677, 2005.

[26] H.-P. Kriegel and M. Pfeifle. Hierarchical density-based clustering of uncertain data. In *IEEE International Conference on Data Mining (ICDM)*, pages 689–692, 2005.

[27] H. W. Kulin and R. E. Kuenne. An efficient algorithm for the numerical solution of the generalized Weber problem in spatial economics. *Journal of Regional Science*, 4(2):21 – 33, 1962.

[28] A. Kumar, Y. Sabharwal, and S. Sen. Linear-time approximation schemes for clustering problems in any dimensions. *Journal of the ACM*, 57(2):1–32, 2010.

[29] C. Lammersen, M. Schmidt, and C. Sohler. Probabilistic k-median clustering in data streams. In *Proceedings of the 10th Workshop on Approximation and Online Algorithms (WAOA)*, 2012.

[30] M. Langberg and L. J. Schulman. Universal epsilon-approximators for integrals. In *Proceedings of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 598–607. Society for Industrial and Applied Mathematics, Austin and Texas, 2010.

[31] S. P. Lloyd. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28(2):129 – 136, 1982.

[32] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004.

[33] N. Megiddo and K. J. Supowit. On the complexity of some common geometric location problems. *SIAM Journal on Computing*, 13(1):182 – 196, 1984.

[34] Z. A. Melzak. *Companion to Concrete Mathematics.* Wiley, 1973.

[35] W. K. Ngai, B. Kao, C. K. Chui, R. Cheng, M. Chau, and K. Y. Yip. Efficient clustering of uncertain data. In *Proceedings 6th IEEE ICDM*, pages 436 – 445, 2006.

[36] E. Weiszfeld. Sur le point pour lequel la somme des distances de n points donnes est minimum. In *Tohoku Math. Journal 43*, volume 43, pages 355–386.

[37] E. Weiszfeld and F. Plastria. On the point for which the sum of the distances to n given points is minimum. *Annals of Operations Research*, 167(1):7–41, 2009.

[38] H. Xu and G. Li. Density-based probabilistic clustering of uncertain data. In *Proceedings 1st CSSE*, volume 4, pages 474 – 477, 2008.

[39] T. Zhang, R. Ramakrishnan, and M. Livny. BIRCH: A New Data Clustering Algorithm and Its Applications. *Data Mining and Knowledge Discovery*, 1(2):141 – 182, 1997.

		Costs			Running Time		
Dataset	k	PROBI		P-LLOYD++	PROBI		P-LLOYD++
BigCross	15	7.55e+08	1.15e+09	—	3.16e+02	3.82e+01	—
	20	7.92e+08	1.19e+09	—	3.62e+02	8.45e+01	—
	25	8.28e+08	9.05e+08	—	4.09e+02	1.73e+02	—
	30	9.40e+08	9.44e+08	—	4.57e+02	2.42e+02	—
	50	8.38e+08	8.30e+08	—	6.39e+02	8.54e+02	—
	100	7.98e+08	7.87e+08	—	1.10e+03	2.82e+03	—
	250	7.21e+08	7.52e+08	—	2.47e+03	8.49e+03	—
Caltech128	10	1.15e+08	1.19e+08	—	1.81e+02	2.22e+01	—
	20	1.11e+08	1.16e+08	—	2.41e+02	1.28e+02	—
	30	1.14e+08	1.15e+08	—	3.00e+02	8.51e+02	—
	40	1.12e+08	1.14e+08	—	3.59e+02	4.41e+02	—
	50	1.13e+08	1.15e+08	—	4.20e+02	2.10e+02	—
	100	1.07e+08	1.13e+08	—	7.14e+02	1.37e+03	—
	150	1.12e+08	1.11e+08	—	9.83e+02	9.76e+03	—
Census	10	1.15e+07	1.15e+07	1.15e+07	7.16e+01	5.94e+01	6.97e+03
	20	1.16e+07	1.15e+07	1.15e+07	9.79e+01	2.83e+02	1.09e+04
	30	1.14e+07	1.15e+07	1.15e+07	1.25e+02	4.34e+02	1.49e+04
	40	1.14e+07	1.15e+07	—	1.52e+02	2.67e+02	—
	50	1.14e+07	1.15e+07	1.15e+07	1.74e+02	1.23e+03	2.29e+04
	75	1.15e+07	1.15e+07	—	2.49e+02	1.64e+03	—
	100	1.15e+07	1.15e+07	—	2.89e+02	4.28e+03	—
Covertype	10	3.42e+07	5.79e+07	4.32e+07	1.31e+01	1.81e+01	1.01e+03
	20	2.23e+07	6.06e+07	3.53e+07	1.74e+01	1.42e+01	2.19e+03
	30	2.32e+07	5.02e+07	3.20e+07	2.12e+01	1.47e+02	2.97e+03
Tower	20	5.08e+06	1.05e+07	—	2.28e+01	1.55e+01	—
	40	6.37e+06	7.99e+06	—	2.92e+01	5.01e+01	—
	60	3.41e+06	8.81e+06	—	3.72e+01	6.77e+01	—
	80	1.96e+06	1.67e+07	—	4.59e+01	2.73e+01	—
	100	2.89e+06	7.96e+06	—	5.23e+01	1.84e+02	—
	250	2.08e+06	9.16e+06	—	1.06e+02	1.05e+03	—

Table 2: Mean values of 100 runs on all tested data sets and k. The first cost column for PROBI gives the cost on the summary, the second gives the cost on the full data set. The first PROBI column in the running time part gives the running time of PROBI, the second indicates the additional time needed by P-LLOYD++ to compute the solution on the summary. The previous diagrams only show the PROBI time.

		Costs			Running Time		
Dataset	k	PROBI		P-LLOYD++	PROBI		P-LLOYD++
BigCross	15	7.55e+08	1.16e+09	—	3.15e+02	3.79e+01	—
	20	7.92e+08	1.19e+09	—	3.62e+02	8.47e+01	—
	25	8.28e+08	9.05e+08	—	4.08e+02	1.71e+02	—
	30	9.40e+08	9.43e+08	—	4.57e+02	2.46e+02	—
	50	8.36e+08	8.30e+08	—	6.38e+02	8.57e+02	—
	100	7.98e+08	7.87e+08	—	1.10e+03	2.82e+03	—
	250	7.21e+08	7.52e+08	—	2.45e+03	8.47e+03	—
Caltech128	10	1.15e+08	1.19e+08	—	1.81e+02	2.17e+01	—
	20	1.11e+08	1.16e+08	—	2.41e+02	1.28e+02	—
	30	1.14e+08	1.15e+08	—	2.99e+02	8.60e+02	—
	40	1.12e+08	1.15e+08	—	3.59e+02	4.40e+02	—
	50	1.13e+08	1.15e+08	—	4.20e+02	2.05e+02	—
	100	1.07e+08	1.13e+08	—	7.14e+02	1.37e+03	—
	150	1.12e+08	1.11e+08	—	9.83e+02	9.78e+03	—
Census	10	1.15e+07	1.15e+07	1.15e+07	7.15e+01	5.74e+01	6.94e+03
	20	1.16e+07	1.15e+07	1.15e+07	9.78e+01	2.76e+02	1.09e+04
	30	1.14e+07	1.15e+07	1.15e+07	1.25e+02	4.49e+02	1.48e+04
	40	1.14e+07	1.15e+07		1.52e+02	2.75e+02	
	50	1.14e+07	1.15e+07	1.15e+07	1.74e+02	1.22e+03	2.27e+04
	75	1.15e+07	1.15e+07		2.44e+02	1.64e+03	
	100	1.15e+07	1.15e+07		2.89e+02	4.28e+03	
Covertype	10	3.43e+07	5.79e+07	4.32e+07	1.31e+01	1.72e+01	1.01e+03
	20	2.23e+07	6.07e+07	3.54e+07	1.73e+01	1.43e+01	2.11e+03
	30	2.32e+07	5.02e+07	3.20e+07	2.12e+01	1.45e+02	2.92e+03
Tower	20	5.04e+06	1.02e+07	—	2.22e+01	1.56e+01	—
	40	6.30e+06	7.99e+06	—	2.92e+01	5.00e+01	—
	60	3.41e+06	8.87e+06	—	3.72e+01	6.76e+01	—
	80	1.95e+06	1.73e+07	—	4.58e+01	2.73e+01	—
	100	2.90e+06	7.80e+06	—	5.22e+01	1.84e+02	—
	250	2.08e+06	9.16e+06	—	1.06e+02	1.04e+03	—

Table 3: Median values of 100 runs on all tested data sets and k. The first cost column for PROBI gives the cost on the summary, the second gives the cost on the full data set. The first PROBI column in the running time part gives the running time of PROBI, the second indicates the additional time needed by P-LLOYD++ to compute the solution on the summary. The previous diagrams only show the PROBI time.

		Costs			Running Time		
Dataset	k	PROBI		P-LLOYD++	PROBI		P-LLOYD++
BigCross	15	0.0155	0.0312	—	0.0092	0.1509	—
	20	0.0095	0.0067	—	0.0020	0.1162	—
	25	0.0060	0.0037	—	0.0032	0.1514	—
	30	0.0101	0.0037	—	0.0023	0.0545	—
	50	0.0110	0.0021	—	0.0031	0.0204	—
	100	0.0074	0.0017	—	0.0016	0.0082	—
	250	0.0058	0.0016	—	0.0153	0.0128	—
Caltech128	10	0.0038	0.0009	—	0.0056	0.1411	—
	20	0.0036	0.0006	—	0.0037	0.1210	—
	30	0.0019	0.0003	—	0.0051	0.0598	—
	40	0.0012	0.0004	—	0.0023	0.1311	—
	50	0.0016	0.0002	—	0.0025	0.1001	—
	100	0.0017	0.0003	—	0.0015	0.0993	—
	150	0.0009	0.0001	—	0.0016	0.0674	—
Census	10	0.0049	0.0002	0.0000	0.0033	0.2906	0.0170
	20	0.0032	0.0001	0.0002	0.0036	0.2044	0.0152
	30	0.0031	0.0001	0.0003	0.0038	0.1562	0.0131
	40	0.0043	0.0004		0.0018	0.0802	
	50	0.0022	0.0002	0.0003	0.0021	0.0093	0.0169
	75	0.0026	0.0002		0.0320	0.0244	
	100	0.0014	0.0002		0.0036	0.0072	
Covertype	10	0.0224	0.0044	0.0000	0.0059	0.1901	0.0655
	20	0.0240	0.0096	0.0008	0.0079	0.1430	0.0650
	30	0.0126	0.0071	0.0014	0.0051	0.0929	0.0497
Tower	20	0.1050	0.1246	—	0.0471	0.1269	—
	40	0.0579	0.0175	—	0.0043	0.0151	—
	60	0.0712	0.0242	—	0.0050	0.0236	—
	80	0.0598	0.1194	—	0.0045	0.0342	—
	100	0.0988	0.0890	—	0.0074	0.0119	—
	250	0.0218	0.0009	—	0.0765	0.0181	—

Table 4: Variance coefficients of 100 runs on all tested data sets and k. The first cost column for PROBI gives the cost on the summary, the second gives the cost on the full data set. The first PROBI column in the running time part gives the running time of PROBI, the second indicates the additional time needed by P-LLOYD++ to compute the solution on the summary. The previous diagrams only show the PROBI time.