Dissertation

# Algorithms for Dynamic Geometric Data Streams

Dipl.–Math. Gereon Frahling

Fakultät für Elektrotechnik, Informatik und Mathematik
Institut für Informatik & Heinz Nixdorf Institut (HNI) &
Paderborn Institute for Scientific Computation (PaSCo)
Warburgerstraße 100, D - 33098 Paderborn

Reviewers: Jun. Prof. Dr. Christian Sohler, Universität Paderborn
Prof. Dr. Friedhelm Meyer auf der Heide, Universität Paderborn
Prof. Dr. Stefano Leonardi, Universita di Roma ”La Sapienza”, Italien

# Acknowledgements

First of all I would like to thank my advisor Christian Sohler for his great support. It was not always easy to keep pace with his great ability to find interesting problems and develop new ideas to solve them. During the whole time he gave me the feeling that I can always ask (even stupid) questions and was responsible for the great atmosphere in Paderborn. Without the fun I had at work I would have never been able to develop the results presented in this thesis.

I also benefited a lot from the great experience of my co-advisor Friedhelm Meyer auf der Heide. He gave me the opportunity to come to Paderborn and the freedom to choose a research area to work on. He always lent a sympathetic ear for all kinds of problems. I would also like to thank Friedhelm's whole research group for the nice time in Paderborn.

Then I would like to thank Kristina for her patience, empathy, love, and all the other things that make it so worthwhile to know her. She turned even the most stressful days into a wonderful time.

Finally I would like to thank those whom I owe the most: my parents, Adolf and Margret. They always gave me the feeling to be loved and supported, whatever I am going to do in my life.

# Contents

# 1 Introduction

The increasing inter-connectivity of modern computer systems has led to the phenomenon of massive data sets occuring in the form of *data streams*. Terabytes of Internet traffic is guided along routers having small memory, telecommunication companies collect gigabytes of communication data each day which have to be analyzed automatically. In almost each area of our digital life a huge amount of data is created. Parts of the data are stored in gigantic data centers on thousands of large hard discs for later analysis. But most of the data is created on the fly, never stored anywhere, and forgotten after seconds.

In this thesis we concentrate on the analysis of such elusive data appearing as a data stream. In the data stream model the data arrives one item by one, and can not be stored locally. We try to maintain a small summary or sketch of the data in local memory. This summary should later help us to answer certain kinds of queries about the data.

A big part of this thesis we will concentrate on a very common problem on huge data sets: *Clustering*. The computational task of clustering is to partition a given input into subsets of equal characteristics. These subsets are usually called clusters and ideally consist of similar objects that are dissimilar to objects in other clusters. This way one can use clusters as a coarse representation of the data. We loose the accuracy of the original data set but we reduce the complexity of the data.

Clustering has straightforward applications in data streaming scenarios. First, we are supposed to handle data sets we are not able to store. Reducing the complexity of these data sets using clustering can give us the ability to store the clustered data for later examination. Second, on such huge data sets clustering often is an effective method to understand the structure of the data set at all. It solves the problem not to see the wood for the trees, which often evolves in the analysis of huge data sets.

To cluster a data set one first has to define a distance measure between objects. This is often done in practice by mapping all objects to points in a $d$-dimensional Euclidean space. The distance between the objects can then be measured by the Euclidean distance between points. Finally one can use established clustering objectives like $k$-median or $k$-means and corresponding algorithms to cluster the data.

One of the main results of this thesis is a technique to reduce the complexity of a huge point set to a weighted point set of logarithmic size, called *coreset*. On the small coreset still good approximate clusterings for the whole point set can be computed.

We will show how to maintain a coreset using polylogarithmic memory on dynamic geometric data streams, consisting of insertions and deletions of points. Our algorithm will give us the ability to compute $(1 \pm \epsilon)$-approximate $k$-median, $k$-means, and MaxCut clusterings on dynamic goemetric data streams using polylogarithmic memory. Having a (problem dependent) mapping

of objects to Euclidean points in mind, we can also compute clusterings on dynamic data streams of arbitrary items.

We now give an overview of the results developed in this thesis.

After introducing some notation, the data stream models, and the clustering objectives used throughout the paper in Chapter 2, we start the development of data stream algorithms in Chapter 3. The data streams we consider here follow the *turnstyle model*, i.e. they consist of update operations on a high dimensional vector. We show how to sample sets of indices almost uniformly at random from the support of the current vector. This is equivalent to sampling random elements from a dynamic multiset $P$ given as a data stream of insert and delete operations. The algorithm uses $O(\log^2(UM/\delta))$ memory bits where $U$ denotes the dimension of the vector (resp. the number of possible elements to be included in the set), $M$ an upper bound on each vector component (resp. the multiplicity of single elements), and $\delta$ the desired statistical difference of the sampling to a uniform one. The difficulty here lies in the desired uniformity of the sampling independently of the multiplicity of the elements in $P$. Furthermore, if the current multiset $P$ is small after many insert and many delete operations, we must be able to reconstruct $P$ to get uniform samples.

We apply this sampling technique to point sets in Chapter 4 and present low-storage data structures to sample points from a dynamic geometric data stream consisting of insertions and deletions of points from the $d$-dimensional discrete Euclidean space $\{1, \ldots, \Delta - 1\}^d$. The sampling is done almost uniformly. We also show direct applications of our sampling technique. Let $P$ be the dynamically evolving point set encoded in the stream.

The data structures developed in Section 4.2 maintain $\epsilon$-nets and $\epsilon$-approximations of range spaces of $P$ having small VC-dimension $\mathcal{D}$. The number of memory bits our data structures use is bounded by $\text{poly}(\mathcal{D}, \epsilon^{-1}, \log(\Delta/\delta))$, where $\delta$ is the desired error probability. Although we do not store the whole point set, we can, after passing over the dynamic geometric data stream, approximately answer certain queries on ranges (e.g. about the number of points in a given rectangle) using the statistics we maintain.

Based on a more sophisticated sampling of points and their respective neighbourhood we also present a low storage data structure to approximate the weight of a Euclidean minimum spanning tree of the points in $P$ in Section 4.4.

The results of Chapters 3 and 4 have been published in [G. *Frahling*, P. Indyk and C. Sohler, Sampling in Dynamic Data Streams and Applications, In: *Proceedings of the 21st Annual Symposium on Computational Geometry (SoCG)*, pages 142–149, 2005. Invited to the special issue of SoCG 2005, to appear in *International Journal of Computational Geometry and Applications (IJCGA)*].

The heart of the thesis is Chapter 5. We develop $(1 + \epsilon)$-approximation algorithms for $k$-median, $k$-means, MaxCut, maximum weighted matching (MaxWM), maximum travelling salesperson (MaxTSP), and average distance. Our algorithms compute a small weighted coreset consisting of $O(k \cdot \log n/\epsilon^{O(d)})$ points that approximates the input point set with respect to the considered problem. The coresets can be computed in nearly linear time.

Having a coreset one only needs a fast approximation algorithm for the weighted problem to

compute a solution quickly. In fact, even an exponential algorithm is sometimes feasible as its running time may still be polynomial in $n$. We will use algorithms from [61] to compute $(1+\epsilon)$-approximate solutions for $k$-median and $k$-means on the coreset in $\text{poly}(\log n, \exp(1/\epsilon))$ time. For MaxCut our technique (reducing the complexity of the big point set to a small coreset and then compute an approximate solution on the coreset) will lead the the fastest known PTAS (in terms of $n$) for Euclidean MaxCut. It is presented in Section 5.5.3.

The new coreset method also has the advantage that it does not rely on assumptions on the optimal clustering solution (in constrast to previous approaches like [61]). This helps us to develop the first efficient algorithms to maintain coresets and during the $m$ insert / delete operations of a dynamic geometric data stream. The space used and the update time per insert / delete operation are bounded by $\text{poly}(\epsilon^{-1}, \log m, \log \Delta)$ for constant $k$ and dimension $d$. At each point of time during the dynamic geometric data stream we can efficiently extract a coreset from the summary held in memory and compute an $(1 \pm \epsilon)$-approximate solution for $k$-median, $k$-means, MaxCut, and all other problems. The algorithms are presented in Chapter 6.

Chapters 5 and 6 are based on [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].

In Chapter 7 we will use the coreset technique developed in Chapter 5 to develop an efficient kinetic data structure to maintain a $(1+\epsilon)$-approximate MaxCut clustering of $n$ points moving linearly in $\mathbb{R}^d$. The data structure is able to answer queries of the form *"to which side of the partition belongs query point* $p$*?"* during the whole movement of the points, each query in time polylogarithmical in $n$.

Previously it was not known if a set of $n$ points moving linearly could force $\Omega(n^2)$ updates of a $(1 \pm \epsilon)$-approximate MaxCut solution. Our data structure shows that such effort is not needed: Under linear motion the data structure processes a number of events linear in $n$, each requiring $O(\log^2 n)$ time. A flight plan update can also be performed in small expected time, when it is performed on a point chosen uniformly from the set of points. No efficient kinetic data structures for MaxCut have been known before.

In Chapter 8 we present an efficient implementation of a $k$-means clustering algorithm. Our algorithm is a variant of KMHybrid [83, 104], i.e. it uses a combination of Lloyd-steps and random swaps, but as a novel feature it uses the coreset construction of Chapter 5 to speed up the algorithm. The main strength of the algorithm is that it can quickly determine clusterings of the same point set for many values of $k$. This is necessary in many applications, since, typically, one does not know a good value for $k$ in advance. Once we have clusterings for many different values of $k$ we can determine a good choice of $k$ using a quality measure of clusterings that is independent of $k$, for example the *average silhouette coefficient*. The average silhouette coefficient can be approximated using coresets.

To evaluate the performance of our algorithm we compare it with algorithm KMHybrid [104] on typical 3D data sets for an image compression application and on artificially created instances. Our data sets consist of $300,000$ to $4.9$ million points. We show that our algorithm significantly outperforms KMHybrid on most of these input instances. Additionally, the quality of the solu-

tions computed by our algorithm deviates less than that of KMHybrid.

We also compute clusterings and approximate average silhouette coefficients for each $k$ between 1 and 100 for our input instances and discuss the performance of our algorithm in detail. The description of the algorithm and experimental results have been previously published in [G. *Frahling* and C. Sohler, A Fast k-Means Implementation using Coresets, In:*Proceedings of the 22nd Annual Symposium on Computational Geometry (SoCG)*, pages 135–143, 2006. Invited to the special issue of SoCG 2006, to appear in *International Journal of Computational Geometry and Applications (IJCGA)*].

Chapter 9 concentrates on graphs given as a data stream of edges. We develop space bounded algorithms that with probability at least $1-\delta$ compute a $(1\pm\epsilon)$-approximation of the number of small *motifs* in these graphs. All algorithms are based on random sampling. Our first algorithm does not make any assumptions on the order of edges in the stream and approximates the number of triangles occuring in the input graph. It uses space that is inversely related to the ratio between the number of triangles and the number of triples with at least one edge in the induced subgraph, and uses constant expected processing time per edge.

Our second triangle counting algorithm is designed for incidence streams (all edges incident to the same vertex appear consecutively). It uses space that is inversely related to the ratio between the number of triangles and the number of paths of length two in the graph and also has small expected processing time per edge. These results significantly improve over previous work [117, 81]. We generalize the results to the counting of cliques of size $k$.

Last but not least we present an algorithm to count the number of bipartite cliques ($K_{3,3}$) with three nodes in each partition in directed incidence streams with bounded out-degree of the nodes. The space needed for the approximation is inversly related to the ratio between the number of $K_{3,3}$ and the number of bipartite cliques having one node in the destination partition ($K_{3,1}$).

Since the space complexity of our algorithms depends only on the structure of the input graph and not on the number of nodes, our algorithms scale very well with increasing graph size and provide a basic tool to analyze the structure of large graphs.

The algorithms to count triangle motifs have been published together with some experiments in [L. Buriol, G. *Frahling*, S. Leonardi, A. Marchetti-Spaccamela, and C. Sohler, Counting Triangles in Data Streams, In: *Proceedings of the 25th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, pages 253–262, 2006].

## 1.1 Motivation

A UC Berkeley study of Lyman and Varian [96] reports that in 2002, the most current year for which there are figures, human mankind stored about 5 exabytes of information (that is 5 million terabytes or 37,000 times the information in the Libraries of Congress). This is just the information which is stored. Most of the information created by human mankind, about 18 exabytes by the same study, is created on the fly and then forgotten. Most of this data would be interesting for later examination, but can't be stored because of high data rate or storage capacity limits.

A prominent example for such data is the internet traffic at a backbone router. Assume we want to maintain some statistics about the routed packets. It would be way to costly to store the required information (e.g., source and destination) for every packet routed. It seems to be much more attractive to maintain a small *sketch* (or *synopsis*) of the data seen so far. Such a sketch should contain an approximation of the information we are interested in. This leads us to data streaming algorithms. They process the data one item by one without storing them. After processing the stream they are able to answer certain queries about the data.

There are many other examples of data streaming scenarios: Telephone call network optimization, sensor networks, banking and credit card transactions, peer to peer connection and transmission data, financial stock trade data, etc.

Data streaming algorithms often even have an advantage in environments where huge data sets are actually stored. According to the study cited above about 92% of the stored data is held on hard disks. From these discs the data can be read sequentially at high rate. Unfortunately only the data items below the head(s) of the hard disk can be accessed immidiately. If we decide to read data from another position on the disk, the disk head has to be moved and after the movement we have to wait for the disk to turn to the desired data item. This takes milliseconds of time even on current high speed disks. Real life applications for huge data therefore try to avoid this random access and just read data sequentially.

Data streaming algorithms can be fed with a sequential stream of the data, avoiding random access at all. If for a given problem we are able to find a data streaming algorithm working with limited main memory, we can guarantee not to trigger any hard disk head moves.

**Dynamic Geometric Data Streams.** Let us assume we have a data stream consisting of insertions and deletions of objects into a big set $Q$. To learn something about the structure of the set $Q$ we have to examine the relations of the objects of $Q$ to each other. First questions arise about the *difference of objects*. To answer such questions we first have to define what *difference of objects* means.

A first attempt would be to model complicated distance measures between objects of $Q$. To each pair of objects $(a, b)$ we assign a number $d(a, b)$ measuring the distance between the objects. This attempt would give us a precise modelling of *difference*, but has a major drawback: How can we store the distances between pairs of objects when we are not even able to store the objects themselves?

The only solution to this problem is to provide an implicit distance measure between objects. In practice objects are often mapped to a $d$-dimensional Euclidean space by a mapping $\alpha$. The distance $d(a, b)$ between objects is then measured as the Euclidean distance between $\alpha(a)$ and $\alpha(b)$ and can be computed implicitly. The dynamic data stream of objects then translates naturally into a *dynamic geometric data stream*, consisting of insertions and deletions of points, by assigning $\alpha$ to each object in the stream.

There are even more direct applications of dynamic geometric data streams: Sensor networks, mobile ad-hoc networks, or the analysis of astrophysical data often provide us directly with data streams of positional data, i.e. data streams of points in $\mathbb{R}^d$. For example, in a mobile

ad-hoc network the participants may regularly broadcast updates of their current position. All participants want to maintain information about the distribution of the participants to maintain an efficient communication network. Since mobile devices have usually limited memory it would be nice to do this using only a small amount of space. Of course, one can model an update as deleting the old position and inserting the new one. Therefore, the model of dynamic geometric data streams applies.

In the context of dynamic geometric data streams we have to reduce the complexity of the point sets. We are able just to store some small representation. Using that we want to answer certain queries about the points. In the geometric context one of the most interesting questions arises about the distribution of points. In Section 4.2 we will address queries like "How many points lie in a certain rectangle?" and show how to answer them by maintaining $\epsilon$-nets and $\epsilon$-approximations of points in small space.

In the context of mobile ad-hoc and sensor networks often the question about the efficience of the current communication networks arises. We can see the Euclidean minimum spanning tree as one of the most efficient communication networks, minimizing to the total length of connections between sensors. Estimating the weight of the Euclidean spanning tree of the current network gives us information about the efficience of the current communication structure. We will show how to measure this Euclidean spanning tree weight in Section 4.4.

**Clustering.** The problem of clustering data sets according to some similarity measure belongs to the most extensively studied optimization problems. Clustering often plays the role of the first step to understand huge data sets. Clustered data help to identify big groups of similar items in the data. Furthermore one can easily find similar items of a query item.

Search engines like Ask.com advertise the ability to present websites on one topic as a cluster to the user. Amazon buyers are clustered to find other users having similar interests. There are many more applications in computational biology, machine learning, data mining and pattern recognition. Since the quality of a clustering is rather problem dependent, there is no general clustering algorithm. Consequently, over the years many different clustering algorithms have been developed.

The most prominent and widely used clustering algorithm is *Lloyd's algorithm* sometimes also referred to as *the* $k$*-means algorithm*. This algorithm requires the input set to be a set of points in the $d$-dimensional Euclidean space. Its goal is to find $k$ cluster centers and a partitioning of the points such that the sum of squared distances to the nearest center is minimized. The algorithm is a heuristic that converges to a local optimum. The main benefit of Lloyd's algorithm is its simplicity and its foundation on analysis of variances. Also, it is relatively efficient.

One major drawback of the $k$-means algorithm is that it needs access to the whole point set in each iteration. In the data streaming scenarios described above it can therefore not be applied directly.

We will provide a method to reduce the complexity of the point set $P$ to a weighted *coreset* in Chapter 5. This method is the first to efficiently compute $(1 \pm \epsilon)$-approximate $k$-median, $k$-means and MaxCut-clusterings in dynamic geometric data streams, where deletions of points are allowed (see Chapter 6). The reduction is done using only polylogarithmic space, and is therefore

applicable to huge dynamic geometric data streams. Using a mapping of objects to Euclidean spaces as described above our reduction technique can also be applied to huge dynamic data streams consisting of insertions and deletions of arbitrary objects.

In Chapter 8 we present an efficient implementation of our coreset reduction technique. It shows that even for data sets in main memory our method can be used to accelerate the first steps of the popular $k$-means method and achieve faster convergence. It can also be used in scenarios where the number of clusters is not specified in advance, because it is suitable to quickly compute clusterings for different numbers of $k$.

**Clustering Kinetic Data.** Clustering is also playing a central role in ad-hoc mobile communication and sensor networks, where the underlying communication structures often depend on the proximity or other similarity characteristics of the stations in motion. For example, mobile networks are often organized in hierarchical clusters, where all the stations inside one cluster are in a close proximity and have direct communication. The hierarchy of clusters then induces a tree structure on their leaders which can be used to establish communication (or perform other data management tasks) between different clusters.

Maintaining clusters of mobile nodes is a very challenging task in mobile networks, because of the dynamic character of the moving nodes. Good clustering algorithms should ensure a tradeoff between the quality of the clustering at any given time and its stability and efficiency under motion.

In Chapter 7 we will develop the first kinetic data structure for MaxCut clustering. We assume that $n$ points (the sensors) are moving along linear trajectories. We show that in this scenario we are able to maintain an approximate MaxCut clustering with $\widetilde{O}(n)$ effort during the motion of points. Updates on the point velocities can be handled efficiently.

Our algorithm is based on the coreset construction technique described in Chapter 5. Since the coreset construction generally applies to the $k$-means and $k$-median problems as well, we are confident that our ideas to develop kinetic data structures using coresets can lead to efficient kinetic data structures for $k$-means and $k$-median clustering in the future as well.

**Counting Motifs in Graphs.** Graphs are fundamental structures for modeling complex relationships between data in Web documents, chemical compounds, XML, social networks etc. A basic tool to uncover their structural design principles and to extract relevant information is to mine the most frequent interconnection patterns occurring in the graph. The computation of network indices based on counting the number of certain small subgraphs is a basic tool in the analysis of the structure of large networks.

As an example, the occurrence of a very large number of certain dense subgraphs has been observed in the Webgraph, the graph formed by Web pages and hyperlinked connections [88], in the attempt of tracing the emergence of hidden cyber-communities. A stochastic model of the growth of the Webgraph [89], the "copying model", has then been developed and uses these dense subgraphs as building blocks of the process of network formation.

Another example are large software systems. The simplest way of reusing the software is to duplicate some portion of the program and later adapt it to some specific needs. In [121]

it is argued that the analysis of frequent network motifs in software architectures suggests that duplication and diversification mechanisms are responsible for a significant part of the observed topological features of large software graphs.

Since huge graphs (webgraph crawls for example) do not fit into main memory, we have to find algorithms working on graphs stored on disks. To avoid random access at all we will look into algorithms working on streams of edges.

Recent implementations and experiments suggest that our algorithms are suitable to compute good estimations on the number of triangles of real webgraph crawls in time comparable to the time to read the graph from the hard disc. Some of the experimental results have already been published in [17].

## 1.2 Related Work

We subdivide the related work into the subjects data streams, geometric data streams, clustering, kinetic data structures, and work related to the counting of motifs in graphs.

**Data Streams.** In 1996 Alon, Matias, and Szegedy analyzed data streaming algorithms to approximate the frequency moments [4] and also gave lower bounds on the memory needed for the approximation. The paper laid the foundation for a big amount of research done in the subsequent years. Some areas of interest are the counting of distinct items and estimating frequency moments [4, 22, 40, 46, 57, 77], the counting of frequent items [24, 30] and the computation of histograms [48, 53, 79]. For other work on streaming algorithms in general we refer to the survey by Muthukrishnan [105].

Ganguly, Garofalakis, and Rastogi gave a method to track set expression cardinalities in dynamic data streams [46]. Their method can potentially be altered to provide random samples as done in Chapter 3. However, this was not the purpose of [46] and the alterations needed are not stated in the paper. Cormode and Muthukrishnan[31] developed a sampling technique for dynamic data streams similar to the technique given in Chapter 3. Their result was obtained at the same time independently of our publication [42].

**Geometric Data Streams.** One of the first geometric problems studied on a stream of points was to approximate the diameter of a point set in the plane [36] using $O(1/\epsilon)$ space. Later this problem has also been considered in higher dimensions [74], where an algorithm with space complexity $O(dn^{1/(c^2-1)})$ to maintain a $c$-approximate diameter for $c > \sqrt{2}$ has been obtained. Chan and Sadjad proposed an algorithm to maintain an approximation of the diameter in the sliding window model [33]. In this model one considers an infinite input stream but only the last $n$ elements are relevant (the window). They gave a $(1+\epsilon)$-approximation algorithm for fixed dimension, which stores $O((\frac{1}{\epsilon})^{(d+1)/2} \log \frac{R}{\epsilon})$ points and where $R$ denotes the ratio between the diameter and the smallest pairwise distance between two points in the window.

Cormode and Muthukrishnan introduced the *radial histogram* [29] to approximate different geometric problems in the plane. A radial histogram is a subdivision of the plane given by concentric circles around a center point and halflines starting at the center. This way we can assign

every point in the stream to a cell of the radial histogram. Problems that can be approximated via radial histrograms include the diameter, convex hull, and the furthest neighbor problem. Hershberger and Suri showed how to maintain a set of $2r$ points such that the distance from the true convex hull of the points seen so far is $O(D/r^2)$ where $D$ is the current diameter of the sample set [69].

Agarwal et al. introduced a framework for approximating various extent measures of point sets for constant dimension [2]. Their technique can be used to obtain streaming algorithms for problems like the diameter, width, smallest bounding box, ball, and cylinder of the point set. Chan used this framework to develop improved streaming algorithms (using less space than previous ones) for a number of geometric problems with constant dimension including diameter, width, minimum-radius enclosing cylinder, minimum width enclosing annulus, etc.

Bagchi et al. [9] gave two deterministic streaming algorithms that maintain $\epsilon$-nets and $\epsilon$-approximations under insertions of points. They apply their algorithm to approximate several robust statistics in data streams including Tukey depth, simplicial depth, regression depth, the Thiel-Sen estimator and the least median of squares. Since their algorithm is deterministic it cannot be extended to the dynamic streaming model including deletions. Suri et al. gave both deterministic and randomized algorithms to compute a (weighted) $\epsilon$-approximation for ranges that are axis-aligned boxes [118].

Indyk introduced the model of dynamic geometric data streams used in this thesis [75]. He gave $O(d \cdot \log \Delta)$-approximation algorithms for (the weight of) minimum weighted matching, minimum bichromatic matching and minimum spanning tree. He also showed how to approximate the weight of an optimal solution of the facility location problem within a factor of $O(d \log^2 \Delta)$. For the k-median problem he gave a $(1+\epsilon)$ approximation algorithm with query time $O(\Delta^{kd} \cdot k \cdot \epsilon^{-d-1}(\log \Delta + \frac{1}{\epsilon} \log \frac{1}{\epsilon}))$ (exhaustive search). He also developed a $O(1)$-approximation that needs $O(\Delta^d \cdot k \cdot \epsilon^{-d-1}(\log \Delta + \frac{1}{\epsilon} \log \frac{1}{\epsilon}))$ time to compute an approximation from the maintained data structure. He further gave a $(1+\epsilon, O(\log \Delta(\log \Delta + \log(1/\epsilon)/\epsilon))$-approximation algorithm, i.e. an algorithm that returns $O(\log \Delta(\log \Delta + \log(1/\epsilon)/\epsilon)) \cdot k$ medians whose cost is at most $1+\epsilon$ times the cost of an optimal algorithm. This algorithm has polylogarithmic query time. All of the algorithms given above work using $O((k + \log \Delta + 1/\epsilon)^{O(1)})$ space.

**Minimum Spanning Tree Approximation.** The problem of estimating the weight of a minimum spanning tree has been considered in the context of sublinear time approximation algorithms. The first such algorithm for the minimum spanning tree weight is designed for sparse graphs and computes a $(1+\epsilon)$-approximation [25]. It has a running time of $\widetilde{O}(D \cdot W/\epsilon^2)$ when the edge weights are in a range from 1 to $W$ and the average degree of the input graph is $D$. In the geometric context a $\widetilde{O}(\sqrt{n}/\epsilon^{O(1)})$ time $(1+\epsilon)$-approximation algorithm was given, if the point set can be accessed using certain complex data structures [32]. In the metric case one can compute a $(1+\epsilon)$-approximation of the minimum spanning tree weight in $O(n/\epsilon^{O(1)})$ time [20].

**Clustering.** It is beyond the scope of this section to give a comprehensive overview of the clustering literature. We want to concentrate on results using *coresets* and then give a brief overview of the most important developments with focus on partitioning algorithms. For a more comprehensive overview of the work in clustering we refer to the surveys/books [14, 65, 80].

Har-Peled and Mazumdar gave $(1+\epsilon)$-approximation algorithms for the k-median and k-means problem, when the points are given in a data stream consisting of insertions (and no deletions) [61]. Their algorithm is based on maintaining coresets of logarithmic size. They also mention the extension of their results to the case of dynamic streaming algorithms as an interesting open problem.

Chan used coresets to approximate different geometric problems including diameter and min-volume bounding box [19]. Together with Sadjad he considered the diameter and width of a point set in the sliding window model [20].

In the context of clustering algorithms several other coreset constructions have been developed for the k-median and k-means clustering problem [8, 60]. These coresets found applications in approximation algorithms [8, 60] and clustering of moving data [59]. Also for projective clustering, coresets have been developed [63].

Apart from clustering, coresets have found applications in basic problems in computational geometry, for example, to compute an approximation of the smallest enclosing ball of a point set [7] or to approximate extent measure of point sets [2, 19].

An overview of coreset constructions is given in [3].

The most popular algorithm for the k-means clustering problem is *Lloyd's algorithm* [41, 62, 95, 97]. It is known that this algorithm converges against a local optimum [115]. Recently, a number of very efficient implementations of this algorithm have been developed [5, 82, 83, 108, 109, 110]. These algorithms reduce the time needed to compute the nearest neighbors in a Lloyd's iteration, which is the most time consuming step of the algorithm. Arthur and Vassilvitskii showed that there are instances which require $2^{\Omega(\sqrt{n})}$ iterations [6].

In the Euclidean space there are many $(1+\epsilon)$-approximation algorithms for the k-means clustering problem [8, 38, 61, 71, 90, 99]. Also for the k-means problem in metric spaces efficient constant factor approximation algorithms are known [83, 100].

The quality of random sampling in metric spaces has been analyzed for some clustering problems including the metric and the Euclidean k-median [34, 102]. The analysis can be easily extended to the k-means clustering problem. A testbed for k-means clustering algorithms has been given in [104].

Streaming algorithms for clustering problems have also been considered in the more general metric space setting [23, 54, 101]. The currently best known algorithm for the k-median problem is a $O(1)$-approximation using $O(k \cdot \text{polylog } n)$ space [21].

**MaxCut.** MaxCut is known to be Max-SNP-hard for general graphs. It has a very easy 0.5-approximation algorithm and an exciting 0.87856-approximation algorithm by Goemans and

Williamson [50]. For metric graphs (and hence also for geometric instances), Fernandez de la Vega, Karpinski, and Kenyon [37] designed a PTAS which computes a $(1 \pm \epsilon)$-approximation in time $O(n^2 \cdot 2^{O(1/\epsilon^2)})$. Indyk [72] designed a PTAS for metric graphs having runtime $O(n \cdot \log n \cdot (2^{(1/\epsilon)^{O(1)}} + \log n))$. These are also the best known algorithms for the Euclidean version of MaxCut, for which it is still not known if the problem is NP-hard.

**Kinetic Data Structures.** There has been a lot of work on designing KDS for various clustering problems. For example, there are efficient KDS for the problems of finding (approximately) minimum number of squares (or other geometric objects) that contain all the input points [47, 68] and for the problem of finding $k$ clusters of minimum maximum radius that cover all points [59]; for other examples, see e.g., [1, 15, 56, 68]. However, unlike the MaxCut problem studied in this thesis, the prior work typically has focused on clustering problems in which each clustered set has been defined only by the *inner-cluster* properties. In contrast, the MaxCut is the problem of clustering the input set of points into two sets for which the sum of the *inter-cluster* distances is maximized, that is, for which the dissimilarity between the points in different clusters is maximized. Comparing to the other clustering problems, MaxCut depends more on *global properties* of the input points and as such it requires different (novel) techniques to be solved efficiently.

**Counting Motifs in Graphs.** Recently much attention has been devoted to the analysis of complex networks arising in information systems, software systems, overlay networks etc. Mining the most frequent subgraphs is here aimed to identify the building blocks of universal classes of complex networks [78, 92]. As an example, the occurrence of a very large number of certain dense subgraphs has been observed in the Webgraph, the graph formed by Web pages and hyperlinked connections [88], in the attempt of tracing the emergence of hidden cyber-communities. A stochastic model of the growth of the Webgraph [89], the "copying model", has these dense subgraphs as building blocks of the process of network formation.

D. Coppersmith and S. Winograd showed that the number of triangles in a subgraph can be counted using matrix multiplication [27]. Schank and Wagner [112] give an extensive experimental study of the performance of algorithms for counting and listing triangles in graphs.

Finding frequent graph patterns also finds application to graph databases[123].

Valverde and Sol used the analysis of frequent network motifs in software architectures to argue that network models based on duplication and diversification mechanisms accounts for a significant part of the observed topological features of large software graphs [121].

# 2 Preliminaries

In this chapter we introduce notations and models used throughout the thesis. In Section 2.1 we begin with general notations.

We will then formally introduce different data stream models and complexity measures used for the analysis of data stream algorithms in Section 2.2. After considering general data streams consisting of arbitrary items we will concentrate on point sets and graphs encoded as data streams.

A big part of the thesis introduces new methods to cluster huge point sets into groups of points. In Section 2.3 we will introduce different clustering objectives, i.e. the k-median, k-means, and MaxCut clustering.

## 2.1 General Notations

We begin with some basic notations and definitions.

We define

$$\mathbb{R}^+ := \{x \in \mathbb{R} \mid x > 0\} \ .$$

We use the $\widetilde{O}$-notation to hide polylogarithmic factors. Formally let $f : \mathbb{R} \to \mathbb{R}^+$ and $g : \mathbb{R} \to \mathbb{R}^+$ be functions with positive function values. We use the notation

$$f(n) = \widetilde{O}(g(n))$$

to measure the growth of $f(n)$ with $n \to \infty$, iff there is a constant $k \in \mathbb{N}$ such that

$$f(n) = O(g(n) \cdot \log^k(g(n))) \text{ as } n \to \infty \ .$$

We will use $[n]$ to denote the set $\{0, \ldots, n-1\}$. For $a, b \in \mathbb{R}$ we define

$$(a, b) := \{x \in \mathbb{R} \mid x > a \wedge x < b\}$$

and

$$[a, b] := \{x \in \mathbb{R} \mid x \geq a \wedge x \leq b\} \ .$$

For $x \in \mathbb{R}$ and $\epsilon \in \mathbb{R}^+$ we define

$$(x \pm \epsilon) := [x - \epsilon, x + \epsilon] \ .$$

We define multiplications of intervals $(a, b)$ with positive scalars $x \in \mathbb{R}^+$ as

$$[a, b] \cdot x := x \cdot [a, b] := [a \cdot x, b \cdot x]$$

and multiplications of positive intervals $(a, b)$ and $(c, d)$ having $0 < a < b$ and $0 < c < d$ as:

$$[a, b] \cdot [c, d] := [a \cdot c, b \cdot d] \ .$$

### Euclidean Spaces

The d-dimensional *vector space* $\mathbb{R}^d$ is the set of all d-tupels $(x_1, \ldots, x_d)$ of real numbers $x_1, \ldots, x_d$. The elements of the space are called *vectors* or *points*. For each point $p \in \mathbb{R}^d$ we denote the ith component of the point as $p^{(i)}$, such that $p = (p^{(1)}, p^{(2)}, \ldots, p^{(d)})$. Addition and subtraction of points in $\mathbb{R}^d$ gives again a point in $\mathbb{R}^d$ and is defined component-wise:

$$\forall_{p,q\in\mathbb{R}^d}(p+q)^{(i)} = p^{(i)} + q^{(i)} \ .$$

The multiplication of a scalar $x \in \mathbb{R}$ with a point $p \in \mathbb{R}^d$ gives again a point in $\mathbb{R}^d$ and is defined as

$$(x \cdot p)^{(i)} = x \cdot p^{(i)} \ .$$

Throughout the paper we will always assume that $d$ is a constant. Therefore we will write $O(d) = O(1)$.

If we restrict the coordinates of a point set to a subset $\mathcal{R}$ of $\mathbb{R}$ we write $\mathcal{R}^d$ for the set of d-tupels of numbers from $\mathcal{R}$. In the thesis we will use the special cases $[0,1]^d$ where $\mathcal{R}$ is the compact interval of numbers between 0 and 1, and $[\Delta]^d = \{0, \ldots, \Delta - 1\}^d$ with $\Delta \in \mathbb{N}$.

As distance measure between the points we will often use the Euclidean distance $d : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}^+$ defined as

$$d(p,q) := \sqrt{\sum_{i=1}^{d}(p^{(i)} - q^{(i)})^2} \ .$$

We generalize this definition to sets, i.e.

$$\forall_{p\in\mathbb{R}^d}\forall_{Q\subset\mathbb{R}^d} d(p,Q) = \min_{q\in Q} d(p,q)$$

and

$$\forall_{Q\subset\mathbb{R}^d}\forall_{R\subset\mathbb{R}^d} d(Q,R) = \min_{q\in Q, r\in R} d(q,r) \ .$$

For a finite set $P = \{p_1, \ldots, p_n\}$ of points from $\mathbb{R}^d$ the *center of gravity* $\mu(P)$ is itself a point in $\mathbb{R}^d$ and defined as

$$\mu(P) = (\mu(P)^{(1)}, \ldots, \mu(P)^{(d)}) \quad \text{with} \quad \mu(P)^{(i)} := \frac{1}{n} \cdot \sum_{p\in P} p^{(i)} \ .$$

We also call the center of gravity $\mu(P)$ the *mean of* $P$.

## 2.2 Data Streams

We now define different types of data streams. The first models we present, the cash register and turnstyle models, are very general and encode (huge) multisets of arbitrary items. We will then present special cases of these cash register and turnstyle models we obtain when we consider points in $[\Delta]^d$ as items. The respective data streams are called geometric data streams and dynamic geometric data streams. Finally we will define streams which encode graphs.

### Cash Register and Turnstyle Model

We first define the cash register and turnstyle models of data streams. They are very general and have been subject of many recent theoretical and practical papers. See the survey of Muthukrishnan [105] for a lineup of recent results.

Let $Q$ be a finite set of $U$ items of different kinds. Let $q_0, \ldots, q_{U-1}$ be these items. We assume that $Q$ is a very huge set.

The data streams we consider encode a multiset of items from $Q$ into a data stream. We use a $U$-dimensional vector $x = (x_0, x_1, \ldots, x_{U-1})$ to describe this multiset: $x_i$ signals that we have exactly $x_i$ items of kind $q_i$ in the current multiset. We assume that we always have at most $M-1$ items of the same kind in our multiset, i.e. $x \in [M]^U$.

We assume that at the beginning of the data stream the multiset is empty, i.e. all $U$ components of $x$ are zero.

In the *turnstyle model* the data stream consists of a sequence of $m$ update operations on the vector $x$. Each update operation has the form $\textsc{Update}(i, a)$ with $i \in [U]$ and $a \in \{-M, \ldots, M\}$ and means that we have to add $a$ to $x_i$. All update operations in the stream lead to an $x_i$ value within $\{0, \ldots, M-1\}$. Thus, at any moment, $0 \leq x_i < M$ for $i \in [U]$ (our algorithms will not verify this assumption). An operation $\textsc{Update}(i, a)$ with $a \geq 0$ can be seen as adding $a$ items of kind $q_i$ to our current multiset. An operation $\textsc{Update}(i, -a)$ with $a > 0$ can be seen as deleting $a$ items of kind $q_i$ from our current multiset. This way we have a huge multiset of items encoded in the stream which is dynamically changing.

In the *cash register model* all $\textsc{Update}(i, a)$ operations have a value $a \geq 0$. Therefore we have to deal only with insertions of items into the current multiset, deletions do not occur in the stream.

Our algorithms see the data streams one update operation by one and we cannot store all data items due to memory restrictions. Particularly random access to the data is impossible. At certain times the algorithms are asked to answer certain queries about the current vector $x$ resp. the current multiset. For example in Chapter 3 we show how to return a random element from the current multiset, each element with approximately the same probability independently of it's multiplicity.

Most of our algorithms will output just an approximate solution to a given problem. For example when we consider optimization problems on the current point set, we will only require the algorithm to return a solution having objective function value within $(1 \pm \epsilon) \cdot \mathit{Opt}$, where $\mathit{Opt}$ denotes the optimal objective function value. We call this solution a $(1 \pm \epsilon)$-approximate solution. We will always assume that $\epsilon < 1/100$.

*Space complexity*. We assume that the size $U$ of the set of possible items, the length of the stream $m$, and the maximum number of occurences of one item $M$ can be very huge. In particular it is impossible to store the whole set of items locally.

Therefore the whole data stored by our algorithm should have bit complexity which is at most polylogarithmic in $U$,$m$, and $M$.

*Time complexity*. We measure time complexity in the Real Random Access Machine (Real RAM) model, as usual in computational geometry to avoid calculations bounding numerical errors. However, note that all algorithms could be altered to use only numbers that could be stored using $c = O\left(\log\left(\frac{mUM}{\epsilon}\right)\right)$ bits, where $\epsilon$ is a small value related to the accuracy of our data streaming algorithms. Since all of our geometric algorithms return approximate results, we are confident that all stated computation times can also be proven in a RAM model having $c$ bits per register.

There are two different measures of time complexity. First, in many real world scenarios we have to react very fast to each UPDATE operation. Therefore we are supposed to develop algorithms which spend time polylogarithmic in $U$,$m$, and $M$ to process an UPDATE operation. The polynomial degree should be as small as possible. Second there is the time to answer a query on the current set of items. The time can be considerably longer than the update time, but it should also preferably be polylogarithmic in $U$,$m$, and $M$.

### Dynamic Geometric Data Streams

The model of dynamic geometric data streams has been first defined in [75]. A dynamic geometric data stream consists of $m$ UPDATE operations on a point set $P \in [\Delta]^d$ in a discrete $d$-dimensional space which is initially empty. We always assume that $\Delta > 100$. An UPDATE operation can be an ADD$(p)$ operation of a point $p \in [\Delta]^d$, which inserts $p$ into $P$, or a REMOVE$(p)$ operation, which deletes $p$ from $P$. We assume that no point is inserted when it is already in $P$ and that no point is deleted from $P$ when it is not currently in the set.

The model of dynamic geometric data streams can be seen as a special case of the turnstyle model by setting $U = \Delta^d$ and $M = 2$. We again assume that we only have access to the sequence one operation by one, and are allowed to use a number of memory bits polylogarithmic in $\Delta$ and $m$.

### Adjacency Streams

Recently much effort has been made to investigate the structure of the webgraph. The difficulty here is that it is quite impossible to hold huge structures in main memory. In a first scenario we could have crawls on the webgraph on hard disc, and are able to access this data. However, random access is very slow and has to be avoided. If we find algorithms which deal with the set of edges given as a data stream, we could solve problems on the webgraph only using sequential access to the data.

In another scenario one or more computers just crawl webpages. They read pages, follow links and read the pages they are directed to recursively. This way these crawlers provide us with a stream of edges.

In the *adjacency stream model* the data stream encodes an undirected graph $G = (V, E)$. We assume that we first are given the node set $V$ and that we are able to store it in local memory.

Then we are able to read a data stream of edges sequentially, i.e. one edge by one. We are not allowed to go back in the data stream and read a previous item. We try to develop algorithms which use less memory bits than needed to encode the graph, i.e. $o(|E|)$ memory bits. This model is called *One pass adjacency stream model*. Algorithms working in this model are called *One pass algorithms*.

When the data is stored on hard disk we can do multiple passes over the data by reading them $k$ times sequentially. We then speak of the $k$ *pass adjacency stream model* and $k$ *pass algorithms*.

### Incidence Streams

*Incidence streams* are a special form of adjacency streams, but now the edges have a special kind of ordering. In the incidence model we assume that each edge $(v, w)$ appears twice, once as $(v, w)$ and once as $(w, v)$. The whole set of $2 \cdot |E|$ edges is then presented as a data stream ordered by source nodes. The model is more restricted than the general adjacency model. We will develop algorithms for incidence streams in Section 9.2 having better memory bounds than the algorithms for adjacency streams.

In Section 9.4 we will alter the model a little bit. We will look at directed graphs having bounded out-degree of the nodes. We are again presented a data stream of edges, but this time each (directed) edge appears just once. The edges are ordered by destination nodes.

## 2.3 Clustering

Assume we have a huge set of objects $O$. To understand the structure of the set it is often a good idea to group the objects into clusters, such that that objects in the same cluster are similar to each other and different from objects in other clusters. To do that we first have to define the notion of *different*. Mathematically this means to define a distance function $d : O \times O \to \mathbb{R}^+$ on pairs of objects. Two objects having a large distance are different. The distance function can be defined in various ways. One way is to embed the points in the $d$-dimensional Euclidean space $\mathbb{R}^d$ by an embedding $\alpha : O \to \mathbb{R}^d$. We can then define the distance between objects as the Euclidean distance between the corresponding points:

$$\forall_{a,b \in O} d(a, b) := d(\alpha(a), \alpha(b)) = \sqrt{\sum_{i=1}^{d} (\alpha(a)^{(i)} - \alpha(b)^{(i)})^2} \ .$$

Having such an embedding in mind we concentrate in the following on clustering points in $\mathbb{R}^d$.

Based on the distance measure $d$ between points we now define certain clustering objectives, i.e. what is a good clustering and what is a bad clustering.

For huge point sets often the $k$-median or $k$-means objective functions are used to define good clusterings. They have the advantage that each cluster can be represented by one single cluster center.

The k-median and k-means objectives measure the distance of all points to their corresponding cluster centers in the following way:

### Weighted Euclidean $k$-median clustering.

In the *weighted Euclidean* k*-median clustering problem* we are given a weighted set $P$ of points in the $\mathbb{R}^d$ with weight function $w : P \to \mathbb{R}^+$. The goal is to find a set $C = \{c_1, \ldots, c_k\}$ of $k$ cluster centers in $\mathbb{R}^d$ and a partition of the set $P$ into $k$ clusters $C_1, \ldots, C_k$ such that

$$\mathit{Median}(P, C, C_1, \ldots, C_k) := \sum_{i=1}^{k} \sum_{p \in C_i} w(p) \cdot d(p, c_i)$$

is minimized. In the unweighted version of the problem all point weights are 1.

If the partition $C_1, \ldots, C_k$ relates each point to its nearest cluster center, i.e. if

$$\forall_{p \in P} \forall_i \left( p \in C_i \Rightarrow d(p, c_i) = \min_j d(p, c_j) \right) \quad ,$$

then we shortly write

$$\mathit{Median}(P, C) := \mathit{Median}(P, C, C_1, \ldots, C_k) \quad .$$

### Weighted Euclidean $k$-means clustering.

In the *weighted Euclidean* k*-means clustering problem* we are given a weighted set $P$ of points in the $\mathbb{R}^d$ with weight function $w : P \to \mathbb{R}^+$. The goal is to find a set $C = \{c_1, \ldots, c_k\}$ of $k$ cluster centers in $\mathbb{R}^d$ and a partition of the set $P$ into $k$ clusters $C_1, \ldots, C_k$ such that

$$\mathit{Means}(P, C, C_1, \ldots, C_k) := \sum_{i=1}^{k} \sum_{p \in C_i} w(p) \cdot d(p, c_i)^2$$

is minimized. In the unweighted version of the problem all weights are 1.

If the partition $C_1, \ldots, C_k$ relates each point to its nearest cluster center, i.e. if

$$\forall_{p \in P} \forall_i \left( p \in C_i \Rightarrow d(p, c_i) = \min_j d(p, c_j) \right) \quad ,$$

then we shortly write

$$\mathit{Means}(P, C) := \mathit{Means}(P, C, C_1, \ldots, C_k) \quad .$$

#### Weighted Euclidean MaxCut clustering.

The *Euclidean MaxCut problem* is the classical MaxCut problem on Euclidean graphs. Therefore, the goal is to find a partition of a point set $P$ into two sets $L$ and $R$ such that the weight of the edges of the complete Euclidean

$$Cut(P, L, R) := \sum_{p \in L} \sum_{q \in R} d(p, q)$$

is maximized.

## 2.4 Chernoff Bounds with Limited Independence

We finally want to introduce a variant of Chernoff bounds from [113] we will frequently use in the analysis of the streaming algorithms we develop. In contrast to traditional Chernoff bounds [58] they assume only $k$-wise independence of the underlying random variables. A set of random variables is called $k$-*wise independent* if the random variables in every subset of $k$ variables are independent.

**Theorem 1 (Theorem 5, [113])** . *If $X$ is the sum of $k$-wise independent random variables, each of which is confined to the interval $[0, 1]$ with $\mu = \mathbf{E}[X]$, then:*

- *For $\delta \leq 1$:*
  - *if $k \leq \lfloor \delta^2 \mu e^{-1/3} \rfloor$, then $\mathbf{Pr}[|X - \mu| \geq \delta\mu] \leq e^{-\lfloor k/2 \rfloor}$.*
  - *if $k = \lfloor \delta^2 \mu e^{-1/3} \rfloor$, then $\mathbf{Pr}[|X - \mu| \geq \delta\mu] \leq e^{-\lfloor \delta^2 \mu/3 \rfloor}$.*
- *For $\delta \geq 1$:*
  - *if $k \leq \lfloor \delta \mu e^{-1/3} \rfloor$, then $\mathbf{Pr}[|X - \mu| \geq \delta\mu] \leq e^{-\lfloor k/2 \rfloor}$.*
  - *if $k = \lfloor \delta \mu e^{-1/3} \rfloor$, then $\mathbf{Pr}[|X - \mu| \geq \delta\mu] \leq e^{-\lfloor \delta\mu/3 \rfloor}$.*
- *For $\delta \geq 1$ and $k = \lceil \delta\mu \rceil$:*

$$\mathbf{Pr}[|X - \mu| \geq \delta\mu] \leq e^{\frac{-\delta \ln(1+\delta)\mu}{2}} < e^{\frac{-\delta\mu}{3}}.$$

# 3 Sampling Data Streams

In this chapter we consider the problem to take a random sample from a dynamic multiset of elements. The sampling should be uniform, i.e. each element of the set should be chosen with almost the same probability independent of it's multiplicity. The dynamic multiset is given as a data stream of insert and delete operations. Usually it is represented by a high dimensional vector. The stream is then a stream of update operations on this vector as defined in the *turnstyle model* in Section 2.2.

In this section we present a general sampling technique on turnstyle data streams. We denote that theoretically a similar method can be obtained from the methods Ganguly, Garofalakis, and Rastogi used in [46]. However, they did not state a sampling result as this was not the purpose of [46]. Cormode and Muthukrishnan[31] developed a sampling technique for turnstyle streams similar to the technique given here. Their result was obtained at the same time independently of our publication [42].

Our methods to sample random elements from turnstyle streams will be specialized to dynamic geometric data streams in Chapter 4. We also give some direct applications of the sampling method there. In Chapter 6 we will use the sampling results together with coreset techniques to compute clusterings of dynamic data streams.

We denote by $x \in [M]^U$ the current vector and by $\text{UPDATE}(i, a)$ the update operations of the vector encoded within the stream.

Let $\text{Supp}(x) = \{i \in [U] | x_i > 0\}$ be the support of $x$. We use notation $\|x\|_0 = |\text{Supp}(x)|$. After passing over the sequence of update operations we want to be able to answer queries which ask about a uniformly distributed random element $i$ from $\text{Supp}(x)$. We want to return the index $i$ and the vector component value $x_i$.

Since we are working on a high dimensional vector $x = (x_0, \ldots, x_{U-1})$, we don't want to use $\Theta(U \cdot \log M)$ memory bits needed by a trivial algorithm. Instead we will develop a data structure which accomplishes the task in space complexity polylogarithmic in $M$ *and* $U$.

Our data structure is parametrized by two numbers $\epsilon, \delta > 0$. The operations are as follows:

- $\text{UPDATE}(i, a)$: performs $x_i \leftarrow x_i + a$, where $i \in [U]$, $a \in \{-M \ldots M\}$.
- SAMPLE: returns either a pair $(i, x_i)$ for some $i \in [U]$ or a flag FAIL. The procedure satisfies the following constraints:
  - If a pair $(i, x_i)$ is returned, then $i$ is chosen at random from $\text{Supp}(x)$ such that for any $j \in \text{Supp}(x)$,
    $\mathbf{Pr}[i = j] = \frac{1}{\|x\|_0} \pm \delta$.

– The probability of returning FAIL is at most $\delta$.

Keeping $O(s)$ instances of this data structure it is possible to choose a sample set $S$ of size $s$ (almost) uniformly at random from the non-zero entries of $v$.

Our sample data structure uses two elementary data structures. The first such structure checks whether there is exactly one non-zero entry in $x$. If this is the case the index of the entry and its value it returned. The second data structure approximates the number of non-zero entries in $x$.

## 3.1 The Unique Element (UE) Data Structure

The first elementary data structure called UE checks whether there is exactly one non-zero entry in $x$. The data structure is deterministic. It supports the following operations on a vector $x = (x_0, \ldots, x_{U-1})$ with entries from $[M]$.

- UPDATE$(i, a)$: as above
- REPORT: if $\|x\|_0 \neq 1$, then it returns FAIL. If $\|x\|_0 = 1$, then it returns the unique pair $(i, x_i)$ such that $x_i > 0$.

The data structure keeps three counters $c_0$, $c_1$, and $c_2$ which are initialized to 0. Our UPDATE operation will ensure that $c_j = \sum_{i \in [U]} x_i \cdot i^j$ at any point of time. The operations UPDATE and REPORT are implemented as follows:

| UPDATE$(i, a)$ | REPORT |
|---|---|
| $c_0 = c_0 + a$ | **if** $c_0 \cdot c_2 - c_1^2 \neq 0$ or $c_0 = 0$ **then return** FAIL |
| $c_1 = c_1 + a \cdot i$ | **else** $i \leftarrow c_1/c_0$ |
| $c_2 = c_2 + a \cdot i^2$ | $x_i \leftarrow c_0$ |
| | **return** $(i, x_i)$ |

**Claim 3.1.1** *The data structure UE uses $O(\log(UM))$ bits of space. One* UPDATE *operation needs $O(1)$ time and a* REPORT *query needs $O(1)$ time. UE returns FAIL if and only if $\|x\|_0 \neq 1$. Otherwise, it correctly returns the unique pair $(i, x_i)$ with $x_i \neq 0$.*

**Proof :** The maximum value of the counters $c_0, c_1, c_2$ is $O(U^3 \cdot M)$ and so the counters need $O(\log(UM))$ bits. It remains to prove that the data structure correctly recognizes the case $\|x\|_0 = 1$ and returns the unique pair $(i, x_i)$ with $x_i \neq 0$. From $x_i \geq 0$ for all $i \in [U]$ it follows

that $c_0 = 0$, if and only if $\|x\|_0 = 0$. Furthermore

$$\begin{aligned}
c_0 \cdot c_2 - c_1^2 &= \left(\sum_{i\in[U]} x_i\right) \cdot \sum_{i\in[U]} x_i \cdot i^2 - \left(\sum_{i\in[U]} x_i \cdot i\right)^2 \\
&= \left(\sum_{i\in[U]} x_i\right) \cdot \sum_{i\in[U]} x_i \cdot i^2 - \sum_{i,j\in[U]} \cdot x_i \cdot i \cdot x_j \cdot j \\
&= \sum_{i,j\in[U]} x_i x_j \cdot j^2 - \sum_{i,j\in[U]} x_i \cdot i \cdot x_j \cdot j \\
&= \frac{1}{2} \sum_{i,j\in[U]} x_i x_j \cdot j^2 + \frac{1}{2} \sum_{i,j\in[U]} x_i x_j \cdot i^2 - \sum_{i,j\in[U]} x_i x_j \cdot i \cdot j \\
&= \frac{1}{2} \sum_{i,j\in[U]} x_i x_j (j^2 - 2ij + i^2) \\
&= \frac{1}{2} \sum_{i,j\in[U]} x_i x_j (j - i)^2 \ .
\end{aligned}$$

All summands are zero unless there exist $i, j \in [U]$ with $i \neq j$ and $x_i, x_j > 0$. In the latter case one summand is positive. Hence, $c_0 \cdot c_2 - c_1 > 0$ iff $\|x\|_0 > 1$. The correctness of the data structure follows immediately. □

## 3.2 The Distinct Elements (DE) Data Structure

The data structure supports two operations on a vector $x = (x_0, \ldots, x_{U-1})$ with entries from $[M]$:UPDATE (as above) and REPORT, which with probability $1 - \delta$ returns a value $k \in [0, U]$ such that $\|x\|_0 \leq k \leq (1 + \psi) \cdot \|x\|_0$; the numbers $\delta$ and $\psi$ are parameters.

One can use a data structure from Ganguly, Garofalakis, and Rastogi[46] to solve this problem using $O(1/\psi^2 \cdot \log(1/\psi) \log(1/\delta) \log(U) \log(UM))$ bits of space. An UPDATE operation needs $O(\frac{1}{\epsilon^2} \cdot \log(1/\delta))$ time, a REPORT operation needs $O(\log U)$ time.

## 3.3 A Sample Data Structure using Totally Random Hash Functions

The basic idea behind our data structure is to use a hash function that maps the universe to a smaller space $[2^j]$. The value $2^j$ corresponds to a guess of the number of non-zero entries currently present in vector $x$. Assuming a fully random hash function every non-zero entry is

mapped to 0 with the same probability. Further, the probability that exactly one non-zero entry is mapped to 0 can be checked using our unique elements data structure. If this is the case, our sample data structure returns the corresponding entry (it returns the index and the value of the entry). We now give the procedure in detail.

Our data structure uses hash functions $h_j, j \in [\lceil \log U \rceil + 1]$. Each $h_j$ is of the form $h_j : [U] \to [2^j]$. Initially, we assume that each $h_j$ is a fully random hash function, we relax this assumption later. The value $2^j$ corresponds to the guess that, currently, there are roughly $2^j$ non-zero entries in $x$.

In addition, we use:

- Unique Element data structures $UE_j, j \in [\lceil \log U \rceil + 1]$, and
- A Distinct Elements data structure $DE$, with parameters $\psi = 1$ and error probability parameter $1/22$.

We write $UE_j$.UPDATE resp. $UE_j$.REPORT to denote an UPDATE resp. REPORT operation for data structure $UE_j$.The operations are implemented as follows:

```
UPDATE(i, a)
  for j ∈ [⌈log U⌉ + 1] do
    if h_j(i) = 0 then
      UE_j.UPDATE(i, a)
  DE.UPDATE(i, a)

SAMPLE
  j = ⌈log(DE.REPORT)⌉
  return UE_j.REPORT
```

**Correctness.** Assume that $DE$ is correct. Note that this happens with probability at least $1 - 1/22$. We have $UE_j$.REPORT$\neq$ FAIL, iff $|\mathrm{Supp}(x) \cap h_j^{-1}(0)| = 1$. Since we assume fully random hash functions, the element reported by $UE_j$ is an element chosen uniformly at random from $\mathrm{Supp}(x)$.

It remains to show a lower bound on the probability of $|\mathrm{Supp}(x) \cap h_j^{-1}(0)| = 1$. Denote $S_j = h_j^{-1}(0)$ and $\ell = \|x\|_0$. Because of $j = \lceil \log(DE.\text{REPORT}) \rceil$ and $\|x\|_0 \leq \log(DE.\text{REPORT}) \leq 2\|x\|_0$, we observe $\ell \leq 2^j \leq 4\ell$. Thus

$$\mathbf{Pr}[|S_j \cap \mathrm{Supp}(x)| = 1] = \ell \cdot 2^{-j} \cdot (1 - 2^{-j})^{\ell-1} \geq 1/4 \cdot (1 - 1/\ell)^{\ell-1} \ .$$

The function $(1-1/\ell)^{\ell-1}$ is monotonically decreasing for $\ell \geq 1$ and $\lim_{\ell\to\infty}(1-1/\ell)^{\ell-1} = 1/e$. Therefore we obtain:

$$\mathbf{Pr}[|S_j \cap \mathrm{Supp}(x)| = 1] \geq \frac{1}{4e} \geq \frac{1}{11} \ .$$

Since the error probability in our distinct elements structure is at most $1/22$, we obtain that our algorithm returns with probability $1/11 - 1/22 = 1/22$ a random element.

## 3.4 A Sample Data Structure using Random Number Generators

We will now give two different methods to overcome the assumption of totally random hash functions and achieve space complexity polylogarithmic in $M$ and $U$. First we will present a general approach based on the random number generator of Nisan [106]. The method introduces a small additive error in the probability of each element to be sampled. This error can be translated into a small multiplicative error of $\epsilon$. However, the method is difficult to implement and for many applications a multiplicative error is sufficient. In Section 3.5 we will present a method based on pairwise independent hash functions. It is easy to implement but introduces a multiplicative error in the probability of each element to be sampled.

To replace the assumption of fully random $h_j$ we use the following lemma developed by Indyk [73] which is based on Nisans random number generator [106].

**Lemma 3.4.1** *[73] Consider an algorithm $A$ which, given a stream $S'$ of pairs $(i, a)$, and a function $f : [n'] \times \{0,1\}^{R'} \times \{-M' \ldots M'\} \to \{-M'^{O(1)} \ldots M'^{O(1)}\}$, does the following:*

- *Set $O = 0$; Initialize length-$R'$ chunks $R_0 \ldots R_{[n']}$ of independent random bits*
- *For each new pair $(i, a)$: perform $O = O + f(i, R_i, a)$*
- *Output $A(S') = O$*

*Assume that the function $f(\cdot,\cdot,\cdot)$ is supported with an evaluation algorithm using $O(C + R')$ space and $O(T)$ time. Then there is an algorithm $A'$ producing output $A'(S)$, that uses only $O(C + R' + \log(M'n'))$ bits of storage and $O([C + R' + \log(M'n')]\log(n'R'))$ random bits, such that*

$$\mathbf{Pr}[A(S) \neq A'(S)] \leq 1/n'$$

*over some joint probability space of randomness of $A$ and $A'$. The algorithm $A'$ uses $O(T + \log(n'R'))$ arithmetic operations per each pair $(i, a)$.* □

For each fixed $j$ our algorithm uses the hash function $h_j$ to select a subset $S_j := \{i \in [U] : h_j(i) = 0\}$ of indices. Each index is in the set $S_j$ with probability $1/2^j$. The UE data structure maintains the values $c_0 = \sum_{i \in S_j} x_i$, $c_1 = \sum_{i \in S_j} x_i \cdot i$, and $c_2 = \sum_{i \in S_j} x_i \cdot i^2$.

Instead of using a hash function $h_j$ to select the set $S_j$ we can use chunks $R_0, \ldots, R_{U-1}$ of $\log U$ random bits. We select $S_j$ as

$$S_j := \{i \in [U] : \text{all of the first } j \text{ bits of } R_i \text{ are } 0\} \ .$$

Again each index $i \in [U]$ is selected to be in the set $S_j$ with probability $1/2^j$. An update $(i, a)$ on the UE data structure then simply adds the functions $f_0$ to $c_0$, $f_1$ to $c_1$, and $f_2$ to $c_2$, where

- $f_1(i, R_i, a) := \begin{cases} 0 \text{ iff one of the first } j \text{ bits of } R_i \text{ is } 1 \\ a \text{ iff all of the first } j \text{ bits of } R_i \text{ are } 0 \end{cases}$

- $f_2(i, R_i, a) := \begin{cases} 0 \text{ iff one of the first } j \text{ bits of } R_i \text{ is } 1 \\ a \cdot i \text{ iff all of the first } j \text{ bits of } R_i \text{ are } 0 \end{cases}$
- $f_3(i, R_i, a) := \begin{cases} 0 \text{ iff one of the first } j \text{ bits of } R_i \text{ is } 1 \\ a \cdot i^2 \text{ iff all of the first } j \text{ bits of } R_i \text{ are } 0 \end{cases}$

We use Lemma 3.4.1 and plug in the values $n' = 1/\delta + U$, $M' = U \cdot M$, $R' = \lceil \log(U) \rceil$, and $C = \log(UM)$. We can replace the random bits $R_i$ by $O([C + R' + \log(M'n')] \log(n'R')) = O(\log(UM/\delta) \log(U/\delta))$ random bits. We use the same random bits for all $j$ and get an algorithm having the desired properties and using just $O(\log^2(UM/\delta))$ bits of storage. The distribution changes by less than $\delta$.

**Lemma 3.4.2** *Given a sequence of update operations on a vector $x = (x_0, \ldots, x_{U-1})$ with entries from $[M]$, there is a data structure that with probability $1/22 - \delta$ returns a pair $(i, x_i)$ with $i \in Supp(x)$ and returns a flag FAIL otherwise. The statistical difference from the distribution of the returned pair to a uniform distribution is at most $\delta$, particularly $\mathbf{Pr}[i = j] = 1/\|x\|_0 \pm \delta$ for every $j \in Supp(x)$. The algorithm uses $O\left(\log^2(UM/\delta)\right)$ space.* □

**Theorem 2 (Sampling in data streams.)** *Let be $\delta \leq \frac{1}{44}$. Given a sequence of update operations on a vector $x = (x_0, \ldots, x_{U-1})$ with entries from $[M]$, there is a data structure that with probability $1 - \delta$ returns $s$ pairs $(i_0, x_{i_0}), \ldots, (i_{s-1}, x_{i_{s-1}})$ with $i_j \in Supp(x)$ and returns a flag FAIL otherwise. The returned pairs are independent of each other and may contain duplicates. The statistical difference from the distribution of each returned pair to a uniform distribution is at most $\delta$, particularly for all $j \in Supp(x)$ and all $k \in [s]$ : $\mathbf{Pr}[i_k = j] = \frac{1}{\|x\|_0} \pm \delta$. The algorithm uses $O\left(\left(s + \log(1/\delta)\right) \cdot \log^2(UM/\delta)\right)$ space.*

**Proof :**

We apply Lemma 3.4.2 and invoke $352 \cdot (s + \ln(1/\delta))$ instances of the data structure, each with independent random choices. Let $Y$ denote the random variable for the number of instances that return a random pair. Since $\delta \leq \frac{1}{44}$ the probability of each instance to return a random element is greater than $\frac{1}{44}$. Hence, $\mathbf{E}[Y] \geq 8 \cdot (s + \ln(1/\delta))$ and

$$\mathbf{Pr}[Y < s] \leq \mathbf{Pr}[Y \leq (1 - (1/2))\mathbf{E}[Y]] \leq e^{(1/2)^2 \cdot \mathbf{E}[Y]/2} \leq \delta \ .$$

Therefore with probability at least $1 - \delta$ we have at least $s$ samples. We return the first $s$ samples we obtain from the instances. □

We will now show that we can recover the whole vector $x$, if we spend space slightly larger than $\|x\|_0$. This will be useful in situations when the support of the current vector is very small.

**Corollary 3.4.3** *Given a sequence of update operations on a vector $x = (x_0, \ldots, x_{U-1})$ with entries from $[M]$ and given an oracle which tells us in advance the value of $\|x\|_0$ at the end of the stream, there is a data structure that with probability $1 - \delta$ returns all pairs $(i_0, x_{i_0}), \ldots, (i_{\|x\|_0 - 1}, x_{i_{\|x\|_0 - 1}})$ and returns a flag FAIL otherwise.*

*The algorithm uses*
$O\left(\|x\|_0 \cdot \log \frac{U}{\delta} \cdot \log^2\left(\frac{UM}{\delta}\right)\right)$ *space.*

**Proof :** If $U < 22$ we can store the whole vector using $\log(M)$ space. Otherwise $\frac{1}{2U} \leq \frac{1}{44}$ and we can apply the data structure of Theorem 2 with $s = 2\|x\|_0(\ln\|x\|_0 + \ln(2/\delta))$ and error probability parameter $\frac{\delta}{2U}$. We simply return all distinct samples we obtain.

Let us fix an arbitrary index $j \in \mathrm{Supp}(x)$. We have

$$\mathbf{Pr}[\forall_{k\in[s]} : j \neq i_k] \leq \left(1 - \frac{1-\frac{\delta}{2U}}{\|x\|_0}\right)^s \leq \left(1 - \frac{1}{2\|x\|_0}\right)^s \leq e^{-\ln\|x\|_0 - \ln(2/\delta)} = \frac{\delta}{2\|x\|_0} \ .$$

It follows from the Union Bound:

$$\mathbf{Pr}[\exists_{j\in\mathrm{Supp}(x)}\forall_{k\in[s]} : j \neq i_k] \leq \|x\|_0 \cdot \frac{\delta}{2\|x\|_0} \leq \delta/2 \ .$$

Therefore, the overall probability of failure is at most $\delta$.

The space requirement of the algorithm is:

$$\begin{aligned} & O\left(\left(s + \log\frac{2U}{\delta}\right)\cdot\log^2\left(\frac{UM}{\delta'}\right)\right) \\ = \ & O\left(\left(\|x\|_0\cdot\left(\log\|x\|_0 + \log\frac{1}{\delta}\right) + \log\frac{U}{\delta}\right)\cdot\log^2\left(\frac{UM}{\delta}\right)\right) \\ = \ & O\left(\|x\|_0\cdot\log\frac{U}{\delta}\cdot\log^2\left(\frac{UM}{\delta}\right)\right) \ . \end{aligned}$$

□

A second consequence can be obtained by translating Theorem 2 (which uses independent draws and thus the sample set may contain multiple copies of the same point) to the case of sampling of random subsets.

**Corollary 3.4.4** *Let $s \leq \|x\|_0/2$. Given a sequence of update operations on a vector $x = (x_0, \ldots, x_{U-1})$ with entries from $[M]$, there is a data structure that with probability $1-\delta$ returns a subset $S \subset$ Supp$(x)$ of $s$ indices and all pairs $(i, x_i)$ for $i \in S$ and returns a flag FAIL otherwise. The statistical difference from the distribution of the returned subset to a uniform distribution is at most $\delta$, particularly $\mathbf{Pr}[j \in S] = \frac{s}{\|x\|_0} \pm \delta$ for every $j \in$ Supp$(x)$.*

*The algorithm uses $O\left(\left(s + \log(U/\delta)\right)\cdot\log^2(UM/\delta)\right)$ space.*

**Proof :** Let us first assume that we have a data structure that returns an index distributed exactly uniformly among Supp$(x)$. Then we can select $s' = c\cdot\left(s + \log(1/\delta)\right)$ indices $i'_0, \ldots, i'_{s'}$ from Supp$(x)$ uniformly at random with repetitions, where $c$ is a suitable constant. Since $s \leq \|x\|_0/2$ we know that for each $i'_j$ we have with probability at least $1/2$ an index that is not among the previously chosen indices. And so we get that with probability at least $1 - \delta/2$ we have at least $s$ distinct indices among $i'_0, \ldots, i'_{s'}$ for $c$ large enough. If there are more than $s$ distinct indices among $i'_0, \ldots, i'_{s'}$ we choose $s$ indices uniformly at random from the distinct indices. Clearly, the computed index set is distributed uniformly at random.

We use Theorem 2 to select $s'$ indices almost uniformly at random. It remains to deal with the fact that Theorem 2 does not provide us with an exact uniform distribution. We will use error probability parameter $\delta' = \delta/(s' \cdot U)$ when we apply the theorem. Each time when we choose a random point the statistical difference to the uniform distribution is at most $\delta' \cdot U$. Since we choose $s'$ elements the overall statistical difference of our process to the ideal one described above is at most $\delta' \cdot U \cdot s' = \delta$. Therefore,$\mathbf{Pr}[j \in S] = \frac{s}{\|x\|_0} \pm \delta$ for every $j \in \mathrm{Supp}(x)$ and the overall probability of error is at most $\delta$.

The space needed by the algorithm is

$$\begin{aligned} & O\left(\left(s' + \log\left(\frac{1}{\delta'}\right)\right) \cdot \log^2\left(\frac{UM}{\delta'}\right)\right) \\ = \; & O\left(\left(s + \log\left(\frac{1}{\delta}\right) + \log\left(\frac{s' \cdot U}{\delta}\right)\right) \cdot \log^2\left(\frac{U^2 M(s + \frac{1}{\delta})}{\delta}\right)\right) \\ = \; & O\left(\left(s + \log\left(\frac{U}{\delta}\right)\right) \cdot \log^2\left(\frac{UM}{\delta}\right)\right) \quad . \end{aligned}$$

□

**Lemma 3.4.5** *Given a sequence of update operations on a vector $x = (x_0, \ldots, x_{U-1})$ with entries from $[M]$, there is a data structure that returns all pairs $(i, x_i)$ with $i \in Supp(x)$ when $\|x\|_0 \leq A$ and a flag* FAIL *if $\|x\|_0 > A$. The algorithm works with probability $1 - \delta$ and uses $O\left(A \cdot \log\left(\frac{U}{\delta}\right) \cdot \log^2\left(\frac{UM}{\delta}\right)\right)$ space.*

**Proof :** We start the algorithm of Corollary 3.4.3 with the assumption that $\|x\|_0 = 2A + 2$ and with error probability parameter $\delta' = \delta/2$. We call it *algorithm 1*. In case that $\|x\|_0 \leq 2A + 2$ at the end of the stream, this algorithm reconstructs all pairs $(i, x_i)$ with $i \in \mathrm{Supp}(x)$.

In parallel we start the algorithm of Corollary 3.4.4 with $s = A + 1$ and error probability parameter $\delta' = \delta/2$ and call it *algorithm 2*. If algorithm 2 returns $A + 1$ elements, we return FAIL. If it returns less than $A + 1$ elements, we know that with probability $1 - \delta/2$ we have $\|x\|_0 < 2A + 2$. In that case algorithm 1 provides us with all pairs $(i, x_i)$ with $i \in \mathrm{Supp}(x)$ with probability $1 - \delta/2$. We count the number of pairs. If $\|x\|_0 \leq A$, we return all pairs, otherwise we return FAIL. □

**Remark 3.4.6** *All of the results stated with an additive error of $\delta$ can be transfered to a multiplicative error of $\delta$: When we apply the results with $\delta' = \delta/U$, we get an additive error of $\delta/U$. So each element is sampled with probability $\frac{1}{\|x\|_0} \pm \frac{\delta}{U}$. Since $\frac{1}{\|x\|_0} \geq \frac{1}{U}$ we conclude that each element is sampled with probability $\frac{1}{\|x\|_0} \cdot (1 \pm \delta)$. Since all memory bounds depend only on $O\left(\log \frac{U}{\delta}\right)$ the replacement of $\delta$ by $\frac{\delta}{U}$ does not change the memory bounds.*

## 3.5 A Sample Data Structure using Pairwise Independent Hash Functions

In this section we focus on the development of a sample data structure that only uses pairwise independent hash functions[18] instead of the approach using a pseudo-random generator. We will consider a relative error instead of an additive error.
The basic idea behind the second sample data structure is similar to the structure using totally random hash functions.

Assume we knew the number $n$ of non-zero entries in our vector $x$. Then we could use a hash function $h$ that maps $[U]$ to a space somewhat larger than $n$, say to $[n/\epsilon]$. It is easy to see that such a hash function has $\epsilon n$ collisions in expectation. This means that typically most of the non-zero entries in $x$ do not collide. We choose a value $\alpha \in [n/\epsilon]$ uniformly at random and check using the UE data structure whether exactly one non-zero entry was mapped to $\alpha$. If this is the case we return the corresponding unique pair $(i, x_i)$ with $h(i) = \alpha$ and $x_i \neq 0$.

Since there are only few collisions this probability is close to $n/(n/\epsilon) = \epsilon$. If we keep $O(1/\epsilon)$ instances of the data structure, one of them is likely to return a pair. We will show that the index of the returned pair is almost uniformly distributed among the non-zero entries in $x$. To deal with the fact that we do not know the value $n$ and that it changes over time we follow the approach of Section 3.3. We will keep $\log U$ hash functions $h_j$, $1 \leq j \leq \lceil \log U \rceil + 2$ each corresponding to a guess $n \approx 2^j$. It will suffice to work with a good guess, which can be identified using our DE data structure.

We now give a detailed description of our sample data structure. Our data structure uses $O(\log U \cdot \log(1/\delta)/\epsilon)$ pairwise independent hash functions $h_{j,k}$ with $j \in [\lceil \log U \rceil + 2]$, $k \in [I]$, and number of instances $I = \left\lceil \log_{(1-\epsilon/32)}(\delta/2) \right\rceil = O(\log(1/\delta)/\epsilon))$. Each $h_{j,k}$ is of the form $h_{j,k} : [U] \to [T]$ with $T := \left\lceil 2^{j+1}/\epsilon \right\rceil$ and corresponds to the $k$-th hash function for the $j$-th guess, $n \approx 2^j$. For each hash function we choose a value $\alpha_{j,k}$, $j \in [\lceil \log U \rceil + 2]$, $k \in [I]$, uniformly at random from $[T]$. Additionally, we need UE data structures $UE_{j,k}$ for $j \in [\lceil \log U \rceil + 2]$ and $k \in [I]$. Each of these data structure is supposed to handle a subset of the UPDATE$(i, a)$ operations in the input stream. Namely, data structure $UE_{j,k}$ will process all updates with $h_{j,k}(i) = \alpha_{j,k}$. We also need one instance of the DE data structure using parameters $\psi = 1$ and error probability parameter $\delta' = \delta/2$.

All hash functions, $\alpha_{j,k}$ and $UE_{j,k}$ are initialized during a separate initialization step. We write $UE_{j,k}$.UPDATE to denote an UPDATE operation for data structure $UE_{j,k}$.

The operations UPDATE and SAMPLE are implemented as follows:

```
UPDATE(i, a)
  for j ∈ [⌈log U⌉ + 2] do
    for k ∈ [I] do
      if h_{j,k}(i) = α_{j,k}
        then UE_{j,k}.UPDATE(i, a)
  DE.UPDATE(i, a)
```

```
SAMPLE
  j = ⌈log(DE.REPORT)⌉
  if ∀_k UE_{j,k}.REPORT=FAIL then return FAIL
  k_0 ← min{k | UE_{j,k}.REPORT≠ FAIL}
  return UE_{j,k_0}.REPORT
```

**Lemma 3.5.1** *Given a sequence of update operations on a vector* $x = (x_0, \ldots, x_{U-1})$ *with entries from* $[M]$*, there is a data structure that with probability* $1 - \delta$ *returns a pair* $(i, x_i)$ *with* $i \in Supp(x)$ *such that* $\mathbf{Pr}[i = j] = (1 \pm \epsilon)/\|x\|_0$ *for every* $j \in Supp(x)$*. The algorithm uses* $O\left(\log(UM/\epsilon) \cdot \log U \cdot \log(1/\delta)/\epsilon\right)$ *space. An* UPDATE *operation can be done in* $O(\log U \cdot \log(1/\delta)/\epsilon)$ *time and a* SAMPLE *query can be processed in* $O(\log U + \log(1/\delta)/\epsilon)$ *time.*

**Proof :** Denote $n := |\mathrm{Supp}(x)| = \|x\|_0$ and assume that DE returns a value DE.REPORT having $n \leq \text{DE.REPORT} \leq 2n$, which happens with probability at least $1 - \delta/2$. We choose $j = \lceil \log(\text{DE.REPORT}) \rceil$ such that $n \leq 2^j \leq 4n$.

Let $k \in [I]$ be fixed. For simplicity of notation we define $h = h_{j,k}$ and $\alpha = \alpha_{j,k}$. We say that $l \in [U]$ is *chosen*, if it is the only entry $x_l$ from $x$ with $x_l \neq 0$ and $h(l) = \alpha$.

For fixed $l, m$ the probability over the choice of $h$ for the event $h(l) = h(m)$ is $1/T$ by pairwise independence. Let us fix an $l$ with $x_l \neq 0$. Using the union bound we get

$$\mathbf{Pr}[\exists_m x_m \neq 0 \wedge h(l) = h(m)] \leq (n-1)/T \leq \epsilon \cdot (n-1)/2^{j+1} \leq \epsilon/2$$

Since $\alpha$ is chosen uniformly at random from the target space $[T]$ and independently of the choice of $h$, the events $h(l) = \alpha$ and $\exists_m (x_m \neq 0 \wedge h(l) = h(m))$ are independent. Therefore,

$$\frac{1}{T} \geq \mathbf{Pr}[l \text{ is chosen}] = \mathbf{Pr}[h(l) = \alpha] \cdot (1 - \mathbf{Pr}[\exists_m (x_m \neq 0 \wedge h(l) = h(m))]) \geq \frac{1}{T} \cdot (1 - \epsilon/2) \ .$$

Since these events are disjoint for all $l$ we get

$$\frac{n}{T} \geq \mathbf{Pr}[\text{any element is chosen}] \geq \frac{n}{T} \cdot (1 - \epsilon/2) \ .$$

It follows for each $l$ with $x_l \neq 0$

$$\mathbf{Pr}[l \text{ is chosen} \mid \text{any element is chosen}] \geq \frac{1}{T} \cdot (1 - \epsilon/2) \Big/ \frac{n}{T} = (1 - \epsilon/2) \cdot 1/n$$

and

$$\mathbf{Pr}[l \text{ is chosen} \mid \text{any element is chosen}] \leq \frac{1/T}{n/T \cdot (1 - \epsilon/2)} = \frac{1}{n \cdot (1 - \epsilon/2)} \leq \frac{1}{n}(1 + \epsilon) \ .$$

Thus, when an element is chosen, it is chosen almost uniformly at random from $\mathrm{Supp}(x)$. It remains to show that for at least one $k$ an element is chosen. From the argumentation above it follows:

$$\mathbf{Pr}[\text{any element is chosen}] \geq \frac{n}{T}(1 - \epsilon/2) \geq \frac{\epsilon n}{2^{j+2}}(1 - \frac{\epsilon}{2}) \geq \frac{\epsilon}{16}(1 - \frac{\epsilon}{2}) = (2 - \epsilon)\frac{\epsilon}{32} \geq \frac{\epsilon}{32}$$

Note that our data structure fails exactly when the data structure DE does not work (probability $\delta/2$) or when no element is chosen for all $k$. No element is chosen, when for all $k$ the data structure $UE_{j,k}$ fails. Since the number of instances $I$ is $\left\lceil \log_{(1-\epsilon/32)}(\delta/2) \right\rceil$ we know that the

probability that no element is chosen for any $k$ is at most $(1-\epsilon/32)^I \leq \delta/2$. Hence the overall probability of error is at most $\delta$ and our data structure is correct with probability at least $1-\delta$.

The algorithm uses $O(\log U \cdot \log(1/\delta)/\epsilon)$ hash functions, $\alpha_{j,k}$ values and UE data structures. Each hash function uses $O(\log U)$ space, each $\alpha_{j,k}$ value uses $O(\log(U/\epsilon))$ space and each UE data structures uses $O(\log(UM))$ space. The DE data structure uses $O(\log(1/\delta)\log(U)\log(UM))$ space.

Hence the overall space complexity is $O\left(\log(UM/\epsilon)\cdot\log U\cdot\log(1/\delta)/\epsilon\right)$. We can evaluate the hash function in constant time and obtain the stated running times. □

**Theorem 3 (Sampling in data streams.)** *Given a sequence of update operations on a vector $x=(x_0,\ldots,x_{U-1})$ with entries from $[M]$, there is a data structure that with probability $1-\delta$ returns $s$ pairs $(i_0,x_{i_0}),\ldots,(i_{s-1},x_{i_{s-1}})$ with $i_j\in Supp(x)$ and returns a flag FAIL otherwise. The returned pairs are independent of each other and may contain duplicates. For all $j\in Supp(x)$ and all $k\in[s]$ : $\mathbf{Pr}[i_k=j]=\frac{1\pm\epsilon}{\|x\|_0}$. The algorithm uses*

$$O\left(\left(s+\log(1/\delta)\right)\cdot\log(UM/\epsilon)\cdot\log(U)/\epsilon\right)$$

*space. An* UPDATE *operation can be processed in $O((s+\log(1/\delta))\cdot\log(U)/\epsilon)$ time. A query operation can be processed in $O((s+\log(1/\delta))\cdot(\log U+1/\epsilon))$ time.*

**Proof :**

We apply Lemma 3.5.1 and invoke $16\cdot(s+\ln(1/\delta))$ instances of the data structure, each with independent random choices and error probability parameter $1/2$. Let $Y$ denote the random variable for the number of instances that return a random pair. The probability of each instance to return a random element is greater than $1/2$. Hence, $\mathbf{E}[Y]\geq 8\cdot(s+\ln(1/\delta))$ and

$$\mathbf{Pr}[Y<s]\leq\mathbf{Pr}[Y\leq(1-(1/2))\mathbf{E}[Y]]\leq e^{(1/2)^2\cdot\mathbf{E}[Y]/2}\leq\delta\ .$$

Therefore with probability at least $1-\delta$ we have at least $s$ samples. We return the first $s$ samples we obtain from the instances. □

**Corollary 3.5.2** *Given a sequence of update operations on a vector $x=(x_0,\ldots,x_{U-1})$ with entries from $[M]$ and given an oracle which tells us in advance the value of $\|x\|_0$ at the end of the stream, there is a data structure that with probability $1-\delta$ returns all pairs $(i_0,x_{i_0}),\ldots,(i_{\|x\|_0-1},x_{i_{\|x\|_0-1}})$ and returns a flag FAIL otherwise.*

*The algorithm uses*
$O\left(\|x\|_0\cdot(\log\|x\|_0+\log(1/\delta))\cdot\log(UM)\cdot\log U\right)$ *space.* UPDATE *operations and query operations can be processed in $O\left(\|x\|_0\cdot(\log\|x\|_0+\log(1/\delta))\cdot\log U\right)$ time.*

**Proof :** We apply Theorem 3 with $s=2\|x\|_0(\ln\|x\|_0+\ln(2/\delta))$, $\epsilon=1/2$, and error probability parameter $\delta/2$ and simply return all distinct samples we get.

Let us fix an arbitrary index $j\in\mathrm{Supp}(x)$. We have

$$\mathbf{Pr}[\forall_{k\in[s]}: j\neq i_k]\leq\left(1-\frac{1-\epsilon}{\|x\|_0}\right)^s\leq\left(1-\frac{1}{2\|x\|_0}\right)^s\leq e^{-\ln\|x\|_0-\ln(2/\delta)}=\frac{\delta}{2\|x\|_0}\ .$$

It follows from the Union Bound:

$$\mathbf{Pr}[\exists_{j\in \mathrm{Supp}(x)} \forall_{k\in[s]} : j \neq i_k] \leq \|x\|_0 \cdot \frac{\delta}{2\|x\|_0} \leq \delta/2 \ .$$

Since the sampling fails with a probability at most $\delta/2$, the overall probability of failure is at most $\delta$.

□

A second consequence can be obtained by translating Theorem 3 (which uses independent draws and thus the sample set may contain multiple copies of the same point) to the case of returning subsets.

**Corollary 3.5.3** *Let $s \leq \|x\|_0/2$. Given a sequence of update operations on a vector $x = (x_0, \ldots, x_{U-1})$ with entries from $[M]$, there is a data structure that with probability $1-\delta$ returns a subset $S \subset Supp(x)$ of $s$ indices and all pairs $(i, x_i)$ for $i \in S$ and returns a flag FAIL otherwise. The algorithm uses $O\left(\left(s + \log(1/\delta)\right) \cdot \log(UM) \cdot \log(U)\right)$ space. UPDATE operations and query operations can be processed in $O((s + \log(1/\delta)) \cdot \log(U))$ time.*

**Proof :** We use the data structure of Theorem 3 with parameter $\epsilon = 1/2$ and error probability parameter $\delta' = \delta/2$ to obtain $s' = c \cdot (s + \log(1/\delta))$ indices $i'_0, \ldots, i'_{s'}$ from $\mathrm{Supp}(x)$ uniformly at random with repetitions, where $c$ is a suitable constant.

Since $s \leq \|x\|_0/2$ we know that for each $i'_j$ either we have at least $s$ different indizes previously chosen or $i'_j$ is with probability at least $\frac{1-\epsilon}{\|x\|_0} \cdot (\|x\|_0 - s) \geq 1/4$ an index that is not among the previously chosen indices. We get that with probability at least $1-\delta/2$ we have at least $s$ distinct indices among $i'_0, \ldots, i'_{s'}$ for $c$ large enough. If there are more than $s$ distinct indices among $i'_0, \ldots, i'_s$ we choose arbitrarily $s$ indices from the distinct indices and return them. □

**Lemma 3.5.4** *Given a sequence of update operations on a vector $x = (x_0, \ldots, x_{U-1})$ with entries from $[M]$, there is a data structure that returns all pairs $(i, x_i)$ with $i \in Supp(x)$ when $\|x\|_0 \leq A$ and a flag FAIL if $\|x\|_0 > A$. The algorithm works with probability $1-\delta$ and uses $O\left(A \cdot (\log A + \log(1/\delta)) \cdot \log(UM) \cdot \log U\right)$ space. UPDATE operations and query operations can be processed in $O(A \cdot (\log A + \log(1/\delta)) \cdot \log(U))$ time.*

**Proof :** We start the algorithm of Corollary 3.5.2 with the assumption that $\|x\|_0 = 2A + 2$ and with error probability parameter $\delta' = \delta/2$. We call it *algorithm 1*. In case that $\|x\|_0 \leq 2A + 2$ at the end of the stream, this algorithm reconstructs all pairs $(i, x_i)$ with $i \in \mathrm{Supp}(x)$.

In parallel we start the algorithm of Corollary 3.5.3 with $s = A + 1$ and error probability parameter $\delta' = \delta/2$ and call it *algorithm 2*. If algorithm 2 returns $A + 1$ elements, we return FAIL. If it returns less than $A + 1$ elements, we know that with probability $1 - \delta/2$ we have $\|x\|_0 < 2A + 2$. In that case algorithm 1 provides us with all pairs $(i, x_i)$ with $i \in \mathrm{Supp}(x)$ with probability $1 - \delta/2$. We count the number of pairs. If $\|x\|_0 \leq A$, we return all pairs, otherwise we return FAIL. □

# 4 Sampling Geometric Data Streams and Applications

In this Chapter we will transfer the sampling technique of Section 3.4 to the context of dynamic geometric data streams. It gives us the ability to choose a point almost uniformly at random from the current point set $P$ encoded in a dynamic geometric data stream.

Based on that we will provide randomized streaming algorithms for three well-studied geometric problems over dynamic geometric streams:

1. Maintaining an $\epsilon$-net of $P$; that is, a subset $N \subset P$ such that for any *range* $R$ from a fixed family of ranges of VC dimension $\mathcal{D}$ (e.g., set of all rectangles), we have $|N \cap R| > 0$, if $\frac{|R \cap P|}{|P|} \geq \epsilon$. We show how to maintain an a set of $\widetilde{O}(\frac{\mathcal{D}+\log(1/\delta)}{\epsilon})$ points that with probability $1-\delta$ is an $\epsilon$-net of $P$.

   Our data structure uses $O\left((\frac{\log(1/\delta)}{\epsilon} + \frac{\mathcal{D}}{\epsilon}\log\frac{\mathcal{D}}{\epsilon}) \cdot d^2 \cdot \log^2(\Delta/\delta)\right)$ space.

2. Maintaining an $\epsilon$-approximation of $P$; that is, a subset $A \subset P$, such that for any *range* $R$ from a fixed family of ranges of bounded VC dimension $\mathcal{D}$, we have $\frac{|A \cap R|}{|A|} = \frac{|R \cap P|}{|P|} \pm \epsilon$. In this case our algorithm maintains a set of $\widetilde{O}(\frac{\mathcal{D}+\log(1/\delta)}{\epsilon})$ points that with probability $1-\delta$ is an $\epsilon$-approximation.

   Our data structure uses $O\left(\frac{1}{\epsilon^2}\left(\mathcal{D}\log\frac{\mathcal{D}}{\epsilon} + \log\frac{1}{\delta}\right) d^3 \log^3\frac{\Delta}{\delta}\right)$ space.

   The $\epsilon$-approximations have applications to many problems, including Tukey depth, simplicial depth, regression depth, the Thiel-Sen estimator, and the least median of squares [9].

3. Maintaining a $(1+\epsilon)$-approximation of the cost of minimum weight tree spanning the points in $P$. This quantity in turn enables to achieve constant factor approximations for other problems, such as TSP or Steiner tree cost. Our algorithms use space $O(\log(1/\delta) \cdot (\log\Delta/\epsilon)^{O(d)})$, and is correct with probability $1-\delta$.

Having a random sample of points from the technique developed in the last chapter, the algorithms to maintain $\epsilon$-nets and $\epsilon$-approximations will follow relatively easily from [66] and [119].

To compute the weight of the Euclidean minimum spanning tree our sampling procedure is used in a more subtle way. It is known that the EMST weight can be expressed as a formula depending on the number of connected components in certain subgraphs of the complete Euclidean graph of the current point set [25, 20]. We use an algorithm from [25] to count the number of connected components in these subgraphs. This algorithm is based on a BFS-like procedure starting at a randomly chosen point $p$. The BFS runs for a constant number of rounds only and one can show that it can never leave a ball around $p$ of certain radius. Therefore, it suffices

to maintain a random sample point and all points in a certain radius around this sample point. This task can also be approximately performed by a variant of our sampling routine described in Section 4.3.

## 4.1 Sampling Geometric Data Streams

First, we observe that we can apply our data structure in the setting of dynamic geometric data streams in the following way. We will use $U = \Delta^d$ and $M = 2$,$[M] = \{0, 1\}$. An ADD$(p)$ operation with $p = (p_0, \ldots, p_{d-1})$ is implemented as an UPDATE$(P, 1)$ operation with $P = \sum_j p_0 \cdot \Delta^j$, i.e. by interpreting $p$ as a $\Delta$-ary number with $d$ digits. In a similar way, a REMOVE$(p)$ operation translates to UPDATE$(P, -1)$. Using the SAMPLE procedure we can get a pair $(i, x_i)$ having $x_i = 1$, which can also be re-interpreted as the corresponding unique point $p = (p_0, \ldots, p_{d-1})$ with $i = \sum_j p_0 \cdot \Delta^j$. Thus we can sample a point from the current point set.

We will now translate the results of Section 3.4 to the context of dynamic geometric data streams.

**Theorem 4 (Sampling in geometric data streams.)** *Given a sequence of* ADD *and* REMOVE *operations of points from the discrete space $[\Delta]^d$, there is a data structure that with probability $1-\delta$ returns $s$ points $r_0, \ldots, r_{s-1}$ from the current point set $P = \{p_0, \ldots, p_{n-1}\}$ and a flag* FAIL *otherwise. The returned points are independent of each other and may contain duplicates. The statistical difference from the distribution of each sample point to a uniform distribution is at most $\frac{\delta}{\Delta^d}$, particularly $\mathbf{Pr}[r_i = p_j] = \frac{1}{n} \pm \frac{\delta}{\Delta^d}$ for every $j \in [n]$.*

*The algorithm uses $O\left(\left(s + \log(1/\delta)\right) \cdot d^2 \cdot \log^2(\Delta/\delta)\right)$ space.*

**Proof :** We apply Theorem 2 with $\delta' = \delta/\Delta^d$. □

We remark that Theorem 4 requires that no point in $P$ occurs more than once, i.e. $P$ is not a multiset.

**Corollary 4.1.1** *Given a sequence of* ADD *and* REMOVE *operations of points from the discrete space $[\Delta]^d$, there is a data structure that with probability $1 - \delta$ returns the current point set $P = \{p_0, \ldots, p_{n-1}\}$.*

*The algorithm uses $O\left(nd^3 \log^3\left(\frac{\Delta}{\delta}\right)\right)$ space.*

**Proof :** This corollary follows directly from Corollary 3.4.3. □

**Corollary 4.1.2** *Let $s \leq n/2$. Given a sequence of* ADD *and* REMOVE *operations of points from the discrete space $[\Delta]^d$, there is a data structure that with probability $1 - \delta$ returns a subset $S = \{r_0, \ldots, r_{s-1}\}$ of $s$ points from the current point set $P = \{p_0, \ldots, p_{n-1}\}$. The statistical difference from the distribution of the returned subset to a uniform distribution is at most $\frac{\delta}{\Delta^d}$, particularly $\mathbf{Pr}[p_j \in S] = \frac{s}{n} \pm \frac{\delta}{\Delta^d}$ for every $j \in [n]$. The algorithm uses $O\left(\left(s + d \cdot \log(\Delta/\delta)\right) \cdot d^2 \cdot \log^2(\Delta/\delta)\right)$ space.*

**Proof :** This corollary follows directly from Corollary 3.4.4. □

## 4.2 $\epsilon$-Nets and $\epsilon$-Approximations in Data Streams

A consequence of Theorem 4 is that we can get $\epsilon$-nets and $\epsilon$-approximations of the current point set. We briefly recapitulate the definitions required for $\epsilon$-nets and $\epsilon$-approximations, which can, for example, be found in [52].

**Definition 4.2.1 (Range Spaces)** *Let $X$ be a set of objects and $R$ be a family of subsets of $X$. Then we call the set system $\Sigma = (X, R)$ a* range space. *The elements of $R$ are the* ranges *of $\Sigma$. If $X$ is a finite set, then $\Sigma$ is called a* finite range space.

**Definition 4.2.2 (VC-dimension)** *The* Vapnik-Chervonenkis dimension (VC-dimension) *of a range space $\Sigma = (X, R)$ is the size of the largest subset of $X$ that is* shattered *by $R$. We say that a set $A$ is* shattered *by $R$, if $\{A \cap r | r \in R\} = 2^A$.*

**Definition 4.2.3 ($\epsilon$-nets, $\epsilon$-approximation)** *Let $\Sigma = (X, R)$ be a finite range space. A subset $N \subset X$ is called $\epsilon$-net, if $N \cap r \neq \emptyset$ for every $r \in R$ with $|r| \geq \epsilon|X|$. A subset $A \subseteq X$ is called $\epsilon$-approximation, if for every $r \in R$ we have $\left|\frac{|A \cap r|}{|A|} - \frac{|r|}{|X|}\right| \leq \epsilon$.*

Obviously, an $\epsilon$-approximation is always an $\epsilon$-net, while the contrary is not necessarily true.

**A Data Streaming Algorithm for $\epsilon$-Approximations.** The following theorem by Vapnik and Chervonenkis shows that for any finite range space with constant VC-dimension $\mathcal{D}$ a random sample of size $\widetilde{O}(\frac{\mathcal{D}+\log(1/\delta)}{\epsilon^2})$ is an $\epsilon$-approximation with probability at least $1 - \delta$.

**Theorem 5** *[119] There is a positive constant $c$ such that if $(X, R)$ is any range space of VC-dimension at most $\mathcal{D}$, $A \subset X$ is a finite subset and $\epsilon, \delta > 0$, then a random subset $B$ of cardinality $s$ of $A$ where $s$ is at least the minimum between $|A|$ and*

$$\frac{c}{\epsilon^2} \cdot \left(\mathcal{D} \cdot \log \frac{\mathcal{D}}{\epsilon} + \log \frac{1}{\delta}\right)$$

*is an $\epsilon$-approximation for $A$ with probability at least $1 - \delta$.*

We can now combine Corollary 4.1.2 (or Corollary 4.1.1 in the case that the current point set is small) with Theorem 5 to obtain a data structure that with probability $1 - \delta$ returns an $\epsilon$-approximation of the current point set.

**Theorem 6** *Given a sequence of* ADD *and* REMOVE *operations of points from the discrete space $[\Delta]^d$, there is a data structure that with probability $1 - \delta$ returns a set $A$ of $\widetilde{O}(\frac{\mathcal{D}+\log(1/\delta)}{\epsilon^2})$ points that is an $\epsilon$-approximation of a range space $(X, R)$ with VC-dimension $\mathcal{D}$. The algorithm uses $O\left(\frac{1}{\epsilon^2}\left(\mathcal{D}\log\frac{\mathcal{D}}{\epsilon} + \log\frac{1}{\delta}\right) d^3 \log^3 \frac{\Delta}{\delta}\right)$ space.*

**Proof :** Let $a = \frac{c}{\epsilon^2} \cdot \left(\mathcal{D} \cdot \log \frac{\mathcal{D}}{\epsilon} + \log \frac{3}{\delta}\right)$. By Theorem 5 a sample set of size $a$ is an $\epsilon$-approximation with probability at least $1 - \delta/3$. Let $P = \{p_0, \ldots, p_{n-1}\}$ denote the current point set. We can easily track the size $n$ of $P$ by increasing a counter with every ADD operation and decreasing it with every REMOVE operation. If $n \leq a$ we use the data structure from Corollary 4.1.1 of size $O\left(ad^3 \log^3 \frac{\Delta}{\delta}\right)$ to recover $P$ completely.

If $n > a$ we will use our data structure from Corollary 4.1.2 of size $O\left(ad^3 \log^3 \frac{\Delta}{\delta}\right)$ to obtain a random sample of size $a$. We will use failure parameter $\delta' = \delta/3$. This guarantees that the overall statistical difference from the same process using the uniform distribution is at most $\delta'/\Delta^d \cdot n \leq \delta/3$. Similarly, the data structure fails with probability at most $\delta' \leq \delta/3$. And a set of size $a$ is with probability $1 - \delta/3$ an $\epsilon$-approximation. Summing up the errors we get an $\epsilon$-approximation with probability $1 - \delta$.

The space requirement follows immediately from Corollaries 4.1.2 and 4.1.1 and Theorem 5. □

**A Data Streaming Algorithm for $\epsilon$-Nets.** Haussler and Welzl showed that a random sample of size $\widetilde{O}(\frac{\mathcal{D}+\log(1/\delta)}{\epsilon})$ is an $\epsilon$-net with probability at least $1 - \delta$.

**Theorem 7** *[66] Let $(X, R)$ be a range space of VC-dimension $\mathcal{D}$, let $A$ be a finite subset of $X$ and suppose $0 < \epsilon, \delta < 1$. Let $N$ be a set obtained by $m$ random independent draws from $A$, where*

$$m \geq \max\left(\frac{4}{\epsilon} \log \frac{2}{\delta}, \frac{8 \cdot \mathcal{D}}{\epsilon} \log \frac{8 \cdot \mathcal{D}}{\epsilon}\right)$$

*then $N$ is an $\epsilon$-net for $A$ with probability at least $1 - \delta$.*

Combining Theorem 4 with Theorem 7 we obtain

**Theorem 8** *Given a sequence of* ADD *and* REMOVE *operations of points from the discrete space $[\Delta]^d$, there is a data structure that with probability $1 - \delta$ returns a set $N$ of $\widetilde{O}(\frac{\mathcal{D}+\log(1/\delta)}{\epsilon})$ points that with probability at least $1 - \delta$ is an $\epsilon$-net of a range space $(X, R)$ with VC-dimension $\mathcal{D}$. The algorithm uses $O\left((\frac{\log(1/\delta)}{\epsilon} + \frac{\mathcal{D}}{\epsilon} \log \frac{\mathcal{D}}{\epsilon}) \cdot d^2 \cdot \log^2(\Delta/\delta)\right)$ space.*

**Proof :** We use a random sample (with repetitions) as given by the data structure from Theorem 4. We choose failure parameter $\delta' = \delta/3$. Since the statistical difference from the exact uniform distribution is at most $\delta' = \delta/3$, the failure probability is at most $\delta' = \delta/3$, and a set of size $\max\left(\frac{4}{\epsilon} \log \frac{2}{\delta'}, \frac{8 \cdot \mathcal{D}}{\epsilon} \log \frac{8 \cdot \mathcal{D}}{\epsilon}\right)$ is an $\epsilon$-net with probability at least $1 - \delta' = 1 - \delta/3$, we get that our sample is an $\epsilon$-net with probability at least $1 - \delta$. □

## 4.3 Random Sampling with Neighborhood Information

We now want to develop a more sophisticated sampling strategy. We would like to draw a set of points (almost) uniformly at random and for each point we also would like to know its

neighborhood, for example all points within a distance of at most $z$ or all points in a square with side length $z$ centered at the random point. Formally, we define a neighborhood in the following way.

**Definition 4.3.1 ($V$-neighborhood)** *Let $V = \{v_0, \ldots, v_{Z-1}\}$ denote a set of grid vectors with $v_0 = (0, \ldots, 0)$. We define the* $V$-neighborhood *of a point $p$ to be the set $\mathcal{N}(V, p) = \bigcup_{i \in Z}\{p+v_i\}$. We call $Z = |V|$ the size of the $V$-neighborhood.*

We will typically assume that the size $Z$ of a $V$-neighborhood is small, i.e. polylogarithmic. We show that we are able to get information about the $V$-neighborhood of a sample point. This can be achieved in the following way:

For simplicity we identify $[\Delta]^d$ with $[\Delta^d]$, such that $P \subset \{0, \ldots, \Delta^d - 1\}$ and $V \subset \{0, \ldots, \Delta^d - 1\}$. We use Theorem 2 with $U = \Delta^d$ and $M = 2^Z$ to map the problem from the discrete Euclidean space to a vector problem. We want to maintain the invariant that the vector $x$ represents the neighborhood of each point in the way:

$$\forall_{p \in \Delta^d} \quad x_p = \sum_{j=0}^{Z-1} a_j \cdot 2^j \quad \text{with} \quad a_j = \begin{cases} 1 \text{ iff } p + v_j \in P \\ 0 \text{ iff } p + v_j \notin P \end{cases} \tag{4.1}$$

where $P$ denotes the current point set after insert and delete operations.
Particularly $x_p = 1 \mod 2 \Leftrightarrow p \in P$.

To maintain the invariant we have to translate the insert and delete operations of points into $Z$ update operations in the following way:

$$\begin{array}{rcl} & & \textbf{for all } i \in [Z] \\ \textsc{Add}(p) & \longrightarrow & \quad \textbf{if } p - v_i \in [\Delta^d] \textbf{ then} \\ & & \quad\quad \textsc{Update}(p - v_i, 2^i) \end{array}$$

$$\begin{array}{rcl} & & \textbf{for all } i \in [Z] \\ \textsc{Remove}(p) & \longrightarrow & \quad \textbf{if } p - v_i \in [\Delta^d] \textbf{ then} \\ & & \quad\quad \textsc{Update}(p - v_i, -2^i) \end{array}$$

We have to deal with the fact that the sample data structure used in Theorem 2 samples elements from $\mathrm{Supp}(x)$, which is a larger set than $P$.

**Lemma 4.3.2** $|P| \geq \frac{\|x\|_0}{Z}$.

**Proof :** Let be $i \in \mathrm{Supp}(x)$, so $x_i = 0$. Because of equation (4.1) there must be a *responsible* point $p \in P$ with $i \in \mathcal{N}(p)$. Since $|\mathcal{N}(p)| = Z$, $p$ can be responsible for the positivity of at most $Z$ entries $x_i$. We conclude that there are at most $|P| \cdot Z$ entries $i$ with $x_i \neq 0$. We conclude $|P| \geq \frac{\|x\|_0}{Z}$. □

We apply Theorem 2 with $s' = 16 \cdot Z \cdot (s + \ln(\delta/2))$ and $\delta' = \frac{\delta}{Z \cdot \Delta^d}$ and get a set $S'$ consisting of $s'$ pairs $(i, x_i)$ with $x_i \neq 0$. Starting with an empty sample set $S$ we check for every pair $(i, x_i)$ if $x_i = 1 \mod 2$. If this is the case we add the pair $(i, x_i)$ (containing the sample point $i \in P$) to our sample set $S$. We stop the procedure when $|S| = s$.

We first show that the set $S$ contains at least $s$ sample points with probability $1 - \delta$. Let $Y$ be the random variable for the number of samples having $x_i = 1 \mod 2$ (being sample points from $P$). For each sample $(i, x_i) \in S'$ and each $p \in P$ we have

$$\mathbf{Pr}[i = p] = \frac{1}{\|x\|_0} \pm \delta' \geq \frac{1}{|P| \cdot Z} - \delta' \geq \frac{1}{2 \cdot |P| \cdot Z} \ .$$

It follows that

$$\mathbf{E}[Y] \geq \frac{1}{2 \cdot |P| \cdot Z} \cdot |P| \cdot s' = 8(s + \ln \delta/2) \ .$$

From Chernoff Bounds [58]:

$$\mathbf{Pr}[Y < s] \leq \mathbf{Pr}[Y \leq (1 - \frac{1}{2})\mathbf{E}[Y]] \leq e^{(1/2)^2 \cdot \mathbf{E}[Y]/2} \leq \frac{\delta}{2} \ .$$

Since Theorem 2 gives us $s'$ samples with probability at least $1 - \delta/2$, we conclude that the constructed set $S$ consists of at least $s$ samples $(i, x_i)$ having $x_i = 1 \mod 2$ with probability $1 - \delta$. It remains to show that the sampling is almost uniform.

Let $S = \{(r_1, x_{r_1}), \ldots, (r_s, x_{r_s})\}$ and let $(k, x_k)$ be the first sample returned by the algorithm of Theorem 2. By the method to construct the set $S$ we see that for all $i \in \{1, \ldots, s\}$ and all $p \in P$:

$$\mathbf{Pr}[r_i = p] = \mathbf{Pr}[k = p] \cdot \frac{\|x\|_0}{|P|} = \left(\frac{1}{\|x\|_0} \pm \delta'\right) \cdot \frac{\|x\|_0}{|P|} = \frac{1}{|P|} \pm \frac{\delta}{Z} \cdot \frac{\|x\|_0}{|P| \cdot \Delta^d}$$

Since $|P| \geq \frac{\|x\|_0}{Z}$ by Lemma 4.3.2 we have:

$$\mathbf{Pr}[r_i = p] = \frac{1}{|P|} \pm \frac{\delta}{\Delta^d} \ .$$

The space requirement to apply Theorem 2 with $s' = 16 \cdot Z \cdot (s + \ln(\delta/2))$, $\delta' = \frac{\delta}{Z \cdot \Delta^d}$, $U = \Delta^d$, and $M = 2^Z$ is:

$$\begin{aligned}
& O\left(\left(s' + \log\left(\frac{1}{\delta'}\right)\right) \cdot \log^2\left(\frac{UM}{\delta'}\right)\right) \\
= \ & O\left(\left(Z \cdot \left(s + \log\left(\frac{2}{\delta}\right)\right) + \log\left(\frac{Z\Delta^d}{\delta}\right)\right) \cdot \log^2\left(\frac{\Delta^d \cdot 2^Z \cdot Z \cdot \Delta^d}{\delta}\right)\right) \\
= \ & O\left(s \cdot Z^3 \cdot d^3 \log^3\left(\frac{\Delta}{\delta}\right)\right)
\end{aligned}$$

**Theorem 9** *Let the set $V = \{v_0, \ldots, v_{Z-1}\}$ be fixed. Given a sequence of* ADD *and* REMOVE *operations of points from the discrete space $[\Delta]^d$, there is a data structure that with probability $1 - \delta$ returns $s$ points $r_0, \ldots, r_{s-1}$ from the current point set $P = \{p_0, \ldots, p_{n-1}\}$ such that $\mathbf{Pr}[r_i = p_j] = \frac{1}{n} \pm \frac{\delta}{\Delta^d}$ for every $j \in [n]$. The points are independent of each other and may contain duplicates. Additionally, the algorithm returns the sets $P \cap \mathcal{N}(V, r_i)$ for every $i \in [s]$. The algorithm uses $\widetilde{O}\left(s \cdot Z^3 \cdot d^3 \cdot \log^3(\Delta/\delta)\right)$ space.* □

## 4.4 Estimating the Weight of a Euclidean Minimum Spanning Tree

In this section we will show how to estimate the weight of a Euclidean minimum spanning tree in a dynamic geometric data stream. We denote by $P = \{p_1, \ldots, p_n\}$ the current point set. Further EMST denotes the weight of the Euclidean minimum spanning tree of the current set.

We impose $\log_{1+\epsilon}(\sqrt{d}\Delta)$ square grids over the point space. The side lengths of the grid cells are $\frac{\epsilon \cdot (1+\epsilon)^i}{\sqrt{d}}$ for $0 \leq i \leq \log_{1+\epsilon}(\sqrt{d}\Delta)$. Our algorithm maintains certain statistics of the distribution of points in the grids. We show that these statistics can be used to compute a $(1+\epsilon)$-approximation of the weight EMST.

Our computation is based on a formula from [33] for the value of the minimum spanning tree of an $n$ point metric space. Let $G_P$ denote the complete Euclidean graph of a point set $P$ and $W$ an upper bound on its longest edge. Further let $c_P^{((1+\epsilon)^i)}$ denote the number of connected components in $G_P^{((1+\epsilon)^i)}$, which is the subgraph of $G_P$ containing all edges of length at most $(1+\epsilon)^i$. Under these assumptions we can use the formula from [33]:

$$\frac{1}{1+\epsilon} \cdot \text{EMST} \leq n - W + \epsilon \cdot \sum_{i=0}^{\log_{1+\epsilon} W - 1} (1+\epsilon)^i c_P^{((1+\epsilon)^i)} \leq \text{EMST} \tag{4.2}$$

where $n$ is the number of points in $P$.

Instead of considering the number of connected components in $G_P^{(t)}$ for $t = (1+\epsilon)^i$ we first move all points of $P$ to the centers of a grid of side length $\frac{\epsilon \cdot t}{\sqrt{d}}$. After removing multiplicities we obtain the point set $P^{(t)}$. Then we consider the graph $G^{(t)}$ whose vertex set is $P^{(t)}$ and that contains an edge between two points if their distance is at most $t$. Instead of counting the connected components in $G_P^{(t)}$ we count the connected components in $G^{(t)}$. It follows from Claim 4.4.1 that this only introduces a small error. We denote by $c^{(t)}$ the number of connected components of $G^{(t)}$. Then we get

**Claim 4.4.1**

$$c_P^{(1+\epsilon)^{i+1}} \leq c^{(1+\epsilon)^i} \leq c_P^{(1+\epsilon)^{i-2}}$$

**Proof :** Let us consider two arbitrary points $p, q \in P$ and the centers of their corresponding cells $p', q'$ in the grid graph $G^{((1+\epsilon)^i)}$. Recall that the corresponding grid has side length $\frac{\epsilon \cdot (1+\epsilon)^i}{\sqrt{d}}$.

Thus by moving $p$ and $q$ to the centers of the corresponding grid cells their distance changes by at most $\epsilon \cdot (1+\epsilon)^i$.

Now assume that $p, q$ are in the same connected component in $G_P^{((1+\epsilon)^{i-2})}$. Then they are connected by a path of edges of length at most $(1+\epsilon)^{i-2}$. If we now consider the path of the corresponding centers of grid cells, then any edge of the path has length at most $(1+\epsilon)^{i-2} + \epsilon \cdot (1+\epsilon)^i \leq (1+\epsilon)^i$ . Therefore, $p', q'$ are in the same connected component of the grid cell graph. We conclude $c^{(1+\epsilon)^i} \leq c_P^{(1+\epsilon)^{i-2}}$.

Assume that $p$ and $q$ are in the same connected component of the grid graph $G^{((1+\epsilon)^i)}$. They are connected by a path of edges of length at most $(1+\epsilon)^i$ in the grid graph $G^{((1+\epsilon)^i)}$. After switching to the point graph $G_P$ any edge of the corresponding path has a length of at most $(1+\epsilon)^i + \epsilon(1+\epsilon)^i = (1+\epsilon)^{i+1}$. Therefore $p$ and $q$ are in the same connected component of $G_P^{((1+\epsilon)^{i+1})}$. We conclude $c_P^{(1+\epsilon)^{i+1}} \leq c^{(1+\epsilon)^i}$. □

We denote by $n^{(t)} = |P^{(t)}|$ the number of non-empty grid cells of side length $\frac{\epsilon t}{\sqrt{d}}$. Our algorithm maintains approximations $\widetilde{n}, \widetilde{W}, \widetilde{n}^{(t)}$, and $\widetilde{c}^{(t)}$ (for $t = (1+\epsilon)^0, (1+\epsilon)^1, (1+\epsilon)^2, \ldots$) of the number $n$ of points currently in the set, the diameter $W$, the size $n^{(t)}$ of $P^{(t)}$, and the number $c^{(t)}$ of connected components in $G^{(t)}$, respectively. The approximation is then derived by inserting the maintained approximations into formula 4.2.

In the following we discuss the data structures we need to maintain our approximations.

### 4.4.1 Required Data Structures

**Number of points.** We observe that we can remember the value of $n$ exactly by increasing/decreasing $\widetilde{n}$ in case of an ADD/REMOVE operation.

**Diameter.** We show how to maintain an approximation $\widetilde{W}$ of $W$ with $W \leq \widetilde{W} \leq 4\sqrt{d}W$, where $W$ is the largest distance between two points in the current point set. To do so, we maintain an approximation $\widetilde{W_j}$ of the diameter of the point set in each of the $d$ dimensions with $W_j \leq \widetilde{W_j} \leq 4W_j$, where $W_j$ is the diameter in dimension $j$ for $1 \leq j \leq d$. The maximum of the $\widetilde{W_j}$ is our approximation $\widetilde{W}$.

We maintain the diameter of the point set in dimension $j$ in the following way: For each $i \in \{0, ..., \log \Delta\}$ we introduce two one-dimensional grids $G_{i,1}$ and $G_{i,2}$, each of them having cells of side length $2^i$. $G_{i,2}$ is displaced by $2^{(i-1)}$ against $G_{i,1}$. Let $g_{i,1}$ and $g_{i,2}$ be the number of occupied cells in the grid $G_{i,1}$, $G_{i,2}$, respectively.

We use our Distinct Elements data structure from Section 3.2 to count the number of grid cells containing a point. We only want to distinguish between the case $g_{i,1} = 1$ and $g_{i,1} > 1$ (we assume that there is always at least one point in the set; otherwise the problem becomes trivial).

If there is exactly one point in the current set we have $g_{0,1} = 1$ and $g_{0,2} = 1$ and the diameter is 0. Otherwise, the diameter must be at least 1. Therefore in the finest grids $G_{0,1}$ and $G_{0,2}$ at least two cells are occupied, which means that $g_{0,1} > 1$ and $g_{0,2} > 1$. We now find the smallest value $i$ such that $g_{i+1,1} = 1$ or $g_{i+1,2} = 1$. In this case we know that $W_j \leq 2^{i+1}$.

Since $g_{i,1} > 1$ and $g_{i,2} > 1$, we know that in both grids $G_{i,1}$ and $G_{i,2}$ at least two cells are occupied. This means that the convex hull of the point set contains the border of a cell in both grids $G_{i,1}$ and $G_{i,2}$. Since these cell borders have a distance of at least $2^{i-1}$ we have $W_j \geq 2^{i-1}$. Therefore, we can output $\widetilde{W}_j = 2^{i+1}$ as a 4-approximation of the diameter in dimension $j$.

**Size of $P^{(t)}$.** The problem to find an estimation $\widetilde{n}^{(t)}$ with $(1-\epsilon)n^{(t)} \leq \widetilde{n}^{(t)} \leq n^{(t)}$ is equivalent to maintaining the number of distinct elements in a data stream. This can be seen as follows. Once a point arrives we can determine its grid cell from its position. Thus we can interpret the input stream as a stream of grid cells and we are interested in the number of distinct grid cells. This can be approximated using an instance of the Distinct Elements (DE) data structure of Section 3.2.

**The Sample Set.** To approximate the number of connected components we have to maintain multisets $S^{(t)}$ of points chosen uniformly at random (with repetitions) from $P^{(t)}$. We will use $V_R$ to denote the set of grid vectors of length at most $R$. For each point $p \in S^{(t)}$ we maintain all points in the $V_R$-neighborhood of $p$ for some suitably chosen value $R$. Since our input stream contains ADD and REMOVE operations of points from $P$ rather than $P^{(t)}$ we have to map every point from $P$ to the corresponding point from $P^{(t)}$. This may have the effect that $P^{(t)}$ becomes a multiset although $P$ is not. This is no problem because our procedure from Lemma 3.4.2 samples from the support of the vector (or, in this case from the support of the multiset). Straightforward modifications of Theorem 9 show that we can also maintain the required sets $S^{(t)}$.

Having such a sample and the value $\widetilde{n}^{(t)}$ we can use an algorithm from [25] to obtain the number of connected components with sufficiently small error. This is proven in Section 4.4.2 using a modified analysis that charges the error in the approximation to the weight of the minimum spanning tree. This way we get our estimation $\widetilde{c}^{(t)}$ of $c^{(t)}$.

## 4.4.2 Computing $\widetilde{c}^{(t)}$

In this section we show how to compute our estimator $\widetilde{c}^{(t)}$. To do this we will use our sample set $S^{(t)}$. In the computation of the sample set $S^{(t)}$ we need to specify the value $R$. We choose

$$R := \log(\sqrt{d} \cdot \Delta) 2^{d+4} \sqrt{d} \epsilon^{-2} \cdot t \ .$$

Further we need in the following the value $D := R/t$. Our algorithm for estimating $c^{(t)}$ works as follows. First, we check, if $\widetilde{W} < 4\epsilon t$. If that is the case, $W < 4\epsilon t$ follows and for an arbitrary sample point we know that every point of the current point set is contained in the radius $R$. Therefore, we know the whole graph $G^{(t)}$ and can compute $c^{(t)}$ exactly.

Thus let us assume $\widetilde{W} \geq 4\epsilon t$. In this case our algorithm is essentially similar to the one presented in [25], but our analysis is somewhat different. The difference comes from the special structure of our input graphs $G^{(t)}$. We exploit the lower bound from Lemma 4.4.2 below to relate the error induced by our approximation algorithm to the weight of the EMST.

**Lemma 4.4.2** *If* $\widetilde{W} \geq 4\epsilon t$ *then*

$$EMST \geq \frac{n^{(t)}\epsilon t}{\sqrt{d}2^{d+1}} \ .$$

**Proof :** We distinguish between the case $n^{(t)} \geq 2^{d+1}$ and $n^{(t)} < 2^{d+1}$.
We start with the case $n^{(t)} \geq 2^{d+1}$. In this case we can color the grid cells using $2^d$ colors in such a way that no two adjacent cells have the same color. Since we have $n^{(t)}$ occupied cells there must be one color $c$ which is assigned to at least $\lceil \frac{n^{(t)}}{2^d} \rceil$ occupied cells. Notice that $\lceil \frac{n^{(t)}}{2^d} \rceil \geq 2$. The occupied cells of the same color are pairwise not adjacent, therefore any pair of points that is contained in two distinct of these cells has a distance of at least $\frac{\epsilon \cdot t}{\sqrt{d}}$. We can conclude EMST $\geq \left( \left\lceil \frac{n^{(t)}}{2^d} \right\rceil - 1 \right) \frac{\epsilon t}{\sqrt{d}} \geq \left( \left\lceil \frac{n^{(t)}}{2^d} \right\rceil - \frac{1}{2} \left\lceil \frac{n^{(t)}}{2^d} \right\rceil \right) \frac{\epsilon t}{\sqrt{d}} \geq \frac{n^{(t)}\epsilon t}{\sqrt{d}2^{d+1}}$ .

In the second case we get $W \geq \frac{\epsilon t}{\sqrt{d}} \geq \frac{n^{(t)}\epsilon t}{\sqrt{d}2^{d+1}}$ since $\widetilde{W} \geq 4\epsilon t$. This implies the result. □

We now present a description of our method to estimate $c^{(t)}$. The idea is pick a random set of vertices (with repetition) and start a BFS with a stochastic stopping rule at each vertex $v$ from the sample to determine the size of the connected component of $v$. If the BFS explores the whole connected component we set a corresponding indicator variable $\beta$ to 1 and else to be 0. To implement this algorithm we can use our sample set $S^{(t)}$. The sample set provides a multiset of points from $P^{(t)}$ chosen uniformly at random. It also provides all other points within a distance of at most $R = Dt$. Since we consider only edges of length at most $t$ and since the algorithm below stops exploring a component when it has size $D$ or larger, the BFS cannot reach a point with distance more than $R$ from the starting vertex. Therefore, our sample set $S^{(t)}$ is sufficient for our purposes. We remark that the random points from $S^{(t)}$ are not chosen *exactly* uniformly. We will choose the failure parameter $\delta'$ in the sampling data structure in such a way that the deviation from the uniform distribution is between $(1-\epsilon)$ and $(1+\epsilon)$ times its probability in the uniform distribution (this means, we choose $\delta' = \epsilon \cdot \delta/\Delta^d$ and each point $p \in P^{(t)}$ is chosen with probability $(1 \pm \epsilon) \cdot \frac{1}{n^{(t)}}$). We take care of this fact in the analysis. The algorithm we use is given below.

APPROXCONNECTEDCOMPONENTS$(P, t, \epsilon)$
  Choose $s$ points $q_1, \ldots, q_s \in P^{(t)}$ uniformly at random
  **for each** $q_i$ **do**
    Choose integer $X$ according to distribution Prob$[X \geq k] = 1/k$
    **if** $X \geq D$ **then** $\beta_i = 0$
    **else**
      **if** Connected component of $G^{(t)}$ containing $q_i$ has at most $X$ vertices
        **then** set $\beta_i = 1$
        **else** set $\beta_i = 0$
  Output: $\hat{c}^{(t)} = \frac{\widetilde{n}^{(t)}}{s} \cdot \sum_{i=1}^{s} \beta_i$

Thus, $\beta_i$ is an indicator random variable for the event that the connected component containing $q_i$ has at most $X$ vertices. We first show upper and lower bounds on the expected output value of

the algorithm. Then we compute the variance and use it to show that the output is concentrated around its expectation. We obtain

$$\begin{aligned}
\mathbf{E}[\beta_i] &= \sum_{\substack{\text{conn. comp.}\\ C \text{ in } G^{(t)}}} \mathbf{Pr}[q_i \in C] \cdot \mathbf{Pr}[X \geq |C| \wedge X < D] \\
&\leq \sum_{\substack{\text{conn. comp.}\\ C \text{ in } G^{(t)}}} \mathbf{Pr}[q_i \in C] \cdot \mathbf{Pr}[X \geq |C|] \\
&\leq \sum_{\substack{\text{conn. comp.}\\ C \text{ in } G^{(t)}}} (1+\epsilon) \cdot \frac{|C|}{n^{(t)}} \cdot \frac{1}{|C|} = (1+\epsilon) \cdot \frac{c^{(t)}}{n^{(t)}} \ .
\end{aligned}$$

For the output value

$$\hat{c}^{(t)} = \frac{\widetilde{n}^{(t)}}{s} \cdot \sum_{i=1}^{s} \beta_i$$

of our algorithm we obtain

$$E[\hat{c}^{(t)}] \leq \frac{\widetilde{n}^{(t)}}{n^{(t)}}(1+\epsilon)c^{(t)} \leq (1+\epsilon)c^{(t)} \ . \tag{4.3}$$

From

$$\begin{aligned}
\mathbf{E}[\beta_i] &= \sum_{\substack{\text{conn. comp.}\\ C \text{ in } G^{(t)}}} \mathbf{Pr}[q_i \in C] \cdot \mathbf{Pr}[X \geq |C| \wedge X < D] \\
&\geq \sum_{\substack{\text{conn. comp.}\\ C \text{ in } G^{(t)}}} (1-\epsilon) \cdot \frac{|C|}{n^{(t)}} \cdot \left(\frac{1}{|C|} - \frac{1}{D}\right) \\
&= (1-\epsilon) \cdot \left(\frac{c^{(t)}}{n^{(t)}} - \frac{1}{D}\right)
\end{aligned}$$

and Lemma 4.4.2 we obtain

$$E[\hat{c}^{(t)}] \geq (1-\epsilon) \cdot \frac{\widetilde{n}^{(t)}}{n^{(t)}} \left(c^{(t)} - \frac{n^{(t)}}{D}\right) \geq (1-\epsilon)^2 \left(c^{(t)} - \frac{EMST \cdot 2^{d+1}\sqrt{d}}{\epsilon t D}\right) \tag{4.4}$$

$$\geq (1-2\epsilon) \left(c^{(t)} - \frac{\epsilon \cdot EMST}{8t \log(\sqrt{d} \cdot \Delta)}\right) \ . \tag{4.5}$$

Our next step is to find an upper bound for the variance of $\hat{c}^{(t)}$. Since the $\beta_i$ are $\{0,1\}$ random variables, we get:

$$\mathbf{Var}[\beta_i] \leq E[\beta_i^2] = E[\beta_i] \leq (1+\epsilon) \cdot \frac{c^{(t)}}{n^{(t)}} \ .$$

By mutual independence of the $\beta_i'$s we obtain for the variance of $\hat{c}^{(t)}$ for fixed $\widetilde{n}^{(t)}$:

$$\begin{aligned}
\mathbf{Var}[\hat{c}^{(t)}] &= \mathbf{Var}[\frac{\widetilde{n}^{(t)}}{s} \sum_{i=1}^{s} \beta_i] \\
&= \frac{\left(\widetilde{n}^{(t)}\right)^2}{s^2} \cdot s \cdot \mathbf{Var}[\beta_i] \\
&\leq (1+\epsilon) \cdot \frac{(\widetilde{n}^{(t)})^2}{s} \cdot \frac{c^{(t)}}{n^{(t)}} \\
&\leq (1+\epsilon) \cdot \frac{n^{(t)} c^{(t)}}{s} \ .
\end{aligned}$$

Using (4.3) and (4.5) we obtain

$$|c^{(t)} - E[\hat{c}^{(t)}]| \leq 2\epsilon c^{(t)} + \frac{3\epsilon \cdot \mathrm{EMST}}{8 \cdot t \cdot \log(\sqrt{d} \cdot \Delta)} \ . \tag{4.6}$$

We choose $s$, the number of sample points, as

$$s := (1+\epsilon) \frac{2^{2d+10} \cdot d \cdot \log^2(\sqrt{d} \cdot \Delta) \cdot \log_{1+\epsilon}(\sqrt{d} \cdot \Delta)}{\epsilon^4} = O\left(\frac{\log^3 \Delta}{\epsilon^5}\right) \ .$$

Chebyshev's inequality and Lemma 4.4.2 imply:

$$\begin{aligned}
\mathbf{Pr}\left[\left|\hat{c}^{(t)} - E[\hat{c}^{(t)}]\right| \geq \frac{\epsilon \cdot \mathrm{EMST}}{8 \cdot t \cdot \log(\sqrt{d} \cdot \Delta)}\right] &\leq (1+\epsilon) \cdot \frac{n^{(t)} \cdot c^{(t)}}{s} \cdot \frac{64 \cdot t^2 \cdot \log^2(\sqrt{d} \cdot \Delta)}{\epsilon^2 \cdot \mathrm{EMST}^2} \\
&\leq (1+\epsilon) \cdot \frac{64 \cdot d \cdot 2^{2d+2} \cdot \log^2(\sqrt{d} \cdot \Delta)}{s \cdot \epsilon^4} \\
&\leq \frac{1}{4 \log_{1+\epsilon}(\sqrt{d} \cdot \Delta)} \ .
\end{aligned}$$

Therefore we get together with (4.6):

**Lemma 4.4.3** *With probability* $1 - \frac{1}{4\log_{1+\epsilon}(\sqrt{d} \cdot \Delta)}$ *we have* $|\hat{c}^{(t)} - c^{(t)}| \leq 2\epsilon c^{(t)} + \frac{\epsilon \cdot EMST}{2 \cdot t \cdot \log(\sqrt{d} \cdot \Delta)}$.

It follows from the union bound that with probability at least $3/4$ all $\widetilde{c}^{(t)}$ values satisfy the inequality in Lemma 4.4.3. It remains to sum up the overall error taking into account that we considered connected components of the graph $G^{(t)}$ and not of the corresponding subgraph of $G_P$. Intuitively, the connected component of $G^{(t)}$ are sufficient because in each of the $G^{(t)}$ we moved every point by at most $\epsilon t$ which is small compared to the threshold edge length of $t$.

**Lemma 4.4.4** *Let $\widetilde{M}$ be the output of our algorithm. Then*

$$\left|EMST - \widetilde{M}\right| \leq 69\sqrt{d} \cdot \epsilon \cdot EMST \ .$$

**Proof :** We will first show that our output value

$$\widetilde{M} := n - \widetilde{W} + \epsilon \sum_{i=0}^{\log_{1+\epsilon} \widetilde{W}-1} (1+\epsilon)^i \widetilde{c}^{((1+\epsilon)^i)}$$

is close to

$$M_p := n - \widetilde{W} + \epsilon \sum_{i=0}^{\log_{1+\epsilon} \widetilde{W}-1} (1+\epsilon)^i c_p^{((1+\epsilon)^i)}$$

which is a $(1+\epsilon)$-approximation of the EMST value by equation (4.2). From Lemma 4.4.3 and Claim 4.4.1 it follows that

$$\begin{aligned} (1-2\epsilon)c_p^{((1+\epsilon)^{i+1})} - \frac{\epsilon \cdot \text{EMST}}{2(1+\epsilon)^i \log(\sqrt{d}\Delta)} &\leq \widetilde{c}^{((1+\epsilon)^i)} \\ &\leq (1+2\epsilon)c_p^{((1+\epsilon)^{i-2})} + \frac{\epsilon \cdot \text{EMST}}{2(1+\epsilon)^i \log(\sqrt{d}\Delta)} \end{aligned}$$

holds with probability at least $1 - \frac{1}{4\log_{1+\epsilon}(\sqrt{d}\cdot\Delta)}$. By the union bound and some calculation we get with probability $3/4$:

$$\begin{aligned} \epsilon \cdot \sum_{i=0}^{\log_{1+\epsilon} \widetilde{W}-1} (1+\epsilon)^i \widetilde{c}^{((1+\epsilon)^i)} &\geq \epsilon \cdot (1-2\epsilon) \sum_{i=0}^{\log_{1+\epsilon} \widetilde{W}-1} (1+\epsilon)^i c_p^{((1+\epsilon)^{i+1})} - \frac{1}{2}\epsilon \cdot \text{EMST} \\ &\geq \epsilon \cdot (1-2\epsilon) \sum_{i=1}^{\log_{1+\epsilon} \widetilde{W}} (1+\epsilon)^{i-1} c_p^{((1+\epsilon)^i)} - \epsilon \cdot \text{EMST} \\ &\geq \epsilon \cdot \frac{1-2\epsilon}{1+\epsilon} \sum_{i=0}^{\log_{1+\epsilon} \widetilde{W}-1} (1+\epsilon)^i c_p^{((1+\epsilon)^i)} - \frac{\epsilon(1-2\epsilon)}{1+\epsilon} n - \epsilon \cdot \text{EMST} \\ &\geq (1-4\epsilon) \cdot \epsilon \sum_{i=0}^{\log_{1+\epsilon} \widetilde{W}-1} (1+\epsilon)^i c_p^{((1+\epsilon)^i)} - \epsilon n - \epsilon \cdot \text{EMST} \end{aligned}$$

and

$$
\begin{aligned}
& \epsilon \cdot \sum_{i=0}^{\log_{1+\epsilon} \widetilde{W}-1} (1+\epsilon)^i \widetilde{c}^{((1+\epsilon)^i)} \\
\leq\ & \frac{1}{2}\epsilon \cdot \mathrm{EMST} + \epsilon \sum_{i=-2}^{\log_{1+\epsilon} \widetilde{W}-3} (1+\epsilon)^{i+2}(1+2\epsilon) c_p^{((1+\epsilon)^i)} \\
\leq\ & \epsilon \cdot \mathrm{EMST} + \epsilon(1+\epsilon)^2(1+2\epsilon) \sum_{i=0}^{\log_{1+\epsilon} \widetilde{W}-1} (1+\epsilon)^i \widetilde{c}_p^{((1+\epsilon)^i)} + 2\epsilon(1+\epsilon)(1+2\epsilon)n \\
\leq\ & (1+11\epsilon) \cdot \epsilon \sum_{i=0}^{\log_{1+\epsilon} \widetilde{W}-1} (1+\epsilon)^i c_p^{((1+\epsilon)^i)} + 12\epsilon n + \epsilon \cdot \mathrm{EMST}
\end{aligned}
$$

which gives us (together with $\widetilde{W} \leq 4\sqrt{d} \cdot \mathrm{EMST}$ and $n \leq \mathrm{EMST}$) a bound on the difference of $M_P$ and $\widetilde{M}$:

$$
\begin{aligned}
\left| M_P - \widetilde{M} \right| &\leq 11\epsilon^2 \sum_{i=0}^{\log_{1+\epsilon} \widetilde{W}-1} (1+\epsilon)^i c_p^{((1+\epsilon)^i)} + 12\epsilon n + \epsilon \cdot \mathrm{EMST} \\
&= 11\epsilon(M_P - n + \widetilde{W}) + 12\epsilon n + \epsilon \cdot \mathrm{EMST} \\
&\leq 24\epsilon \cdot \mathrm{EMST} + 44\epsilon\sqrt{d} \cdot \mathrm{EMST} \ .
\end{aligned}
$$

By the triangle inequality and (4.2) we get the final result:

$$
\begin{aligned}
\left| \widetilde{M} - \mathrm{EMST} \right| &\leq \left| \widetilde{M} - M_P \right| + |M_P - \mathrm{EMST}| \leq 24\epsilon \cdot \mathrm{EMST} + 44\epsilon\sqrt{d} \cdot \mathrm{EMST} + \epsilon \cdot \mathrm{EMST} \\
&\leq 69\epsilon\sqrt{d} \cdot \mathrm{EMST} \ .
\end{aligned}
$$

□

From this lemma our final result follows immediately using standard amplification techniques to ensure that the estimation is correct at every point of time.

**Theorem 10** *Given a sequence of insertions / deletions of points from the discrete $d$-dimensional space $\{1, \ldots, \Delta\}^d$ there is a streaming algorithm that uses $O(\log^3(1/\delta) \cdot (\log(\Delta)/\epsilon)^{O(d)})$ space and $O(\log^3(1/\delta) \cdot (\log(\Delta)/\epsilon)^{O(d)})$ time (for constant $d$) for each update and computes with probability $1-\delta$ a $(1+\epsilon)$-approximation of the weight of the Euclidean minimum spanning tree.*
□

# 5 The Coreset Method

In this chapter we introduce a method to reduce the complexity of huge point sets. Assume we are given a huge point set $P$. To understand the structure of the point set it is often a good idea to partition the point set into clusters, such that two points near one another are in the same cluster, but points having a big distance of each other are in different clusters. In this section we will look at different clustering objectives: The k-median, k-means, and MaxCut clustering.

Computing clusterings on the huge point sets directly is often impossible. Traditional clustering algorithms usually have time and space complexity at least linear in the number of points and need random access to the data. For huge point sets which do not fit into our local memory at all traditional algorithms are not applicable.

In this chapter we will address this problem by introducting a new technique to reduce the complexity of the point set. We will combine points to so called *weighted coreset points*. One coreset point $p$ having a weight of $w(p)$ then represents $w(p)$ points of our input instance. We will compute a coreset having certain theoretical guarantees. The most important ones are that it is small (it's size is logarithmic in the number of points) and that each clustering solution computed on the coreset is a $(1 \pm \epsilon)$-approximate solution on the input point set.

Unlike previous coreset construction techniques [8, 60, 61] our method does not make assumptions on the distribution of points in advance. This will enable us to develop the fastest known PTAS for Euclidean MaxCut in Section 5.5.3, the first efficient k-median and k-means clustering algorithms for dynamic geometric data streams in Chapter 6, the first kinetic data structures for MaxCut in Chapter 7 and efficient k-means implementations for huge point sets in Chapter 8.

## 5.1 Definitions

The most important problems we will address with our coreset technique are the clustering problems k-median, k-means, and MaxCut as defined in Section 2.3. However, we will define a set of other example problems here, which can as well be solved using the coreset technique presented later.

The *maximum matching* problem asks to find a perfect matching of the points in $P$ that maximizes the sum of the length of the matching edges. For a matching $\mathcal{M}$ we denote its cost by

$$\mathit{MaxMatching}(P, \mathcal{M}) = \sum_{(p,q)\in\mathcal{M}} d(p, q) \ .$$

The *maximum travelling salesperson problem* is to find a simple cycle $\mathcal{C}$ (a tour) of the points in

$P$ with maximal cost. We denote its cost by

$$\mathit{MaxTSP}(P, \mathcal{C}) = \sum_{(p,q)\in\mathcal{C}} d(p, q) \ .$$

We will also consider the problem to compute the *average distance* between points in $P$.

### 5.1.1 Oblivious Optimization Problems

In this section we define *oblivious optimization problems over point sets*. Intuitively, an oblivious optimization problem has the property that for a fixed input the set of feasible solutions depends on the cardinality of the input set and not on the input set itself. Hence, there is a set of solutions that are feasible independent of the position of the input points. The quality of the solutions, however, may depend on the positions. MaxCut, MaxMatching, MaxTSP, and AverageDistance can be formulated as oblivious optimization problems.

Let us consider an optimization problem $\Pi$ on point sets in the $\mathbb{R}^d$. $\Pi$ can be either a maximization or minimization problem.

We call $\Pi$ an *oblivious optimization problem*, if it has the following structure. For any $n$ tuple of points $P = (p_1, \ldots, p_n)$ let $\mathcal{S}_\Pi(n)$ denote the set of *feasible solutions*, i.e. the set of feasible solutions depends only on the *size* of the input instance and not on the instance itself. Further, we have for each $n \in \mathbb{N}$ and $s \in \mathcal{S}_\Pi(n)$ an objective function $\mathit{cost}_\Pi^{(n,s)}$ that assigns to $P = (p_1, \ldots, p_n)$ a non-negative cost. We assume that given a permutation $\pi$ of the points and a solution $s$, there is always another solution $s'$ having the cost $\mathit{cost}_\Pi^{(n,s')}(\pi(p_1), \ldots, \pi(p_n)) = \mathit{cost}_\Pi^{(n,s)}(p_1, \ldots, p_n)$ (We say that $\Pi$ does not depend on the order of the points). If $\Pi$ is a maximization (minimization) problem then one seeks to find for a given $P = \{p_1, \ldots, p_n\}$ the solution $s^*$ that maximizes (minimizes) $\mathit{cost}_\Pi^{(n,s)}(p_1, \ldots, p_n)$. We write $\mathit{Opt}_\Pi(P) := \mathit{cost}_\Pi^{(n,s^*)}(p_1, \ldots, p_n)$.

**Example 5.1.1 Euclidean MaxCut:** *A feasible solution $s$ consists of a partition of $\{1, \ldots, n\}$ into 2 groups $C_1, C_2$. For technical reasons we scale the usually used cost of the MaxCut problem by $\frac{1}{n}$. The cost of $s$ on $P = \{p_1, \ldots, p_n\}$ is then given by*

$$\mathit{cost}_{\mathit{maxcut}}^{(n,s)}(p_1, \ldots, p_n) = \frac{1}{n} \sum_{i\in C_1} \sum_{j\in C_2} d(p_i, p_j) \ .$$

**Example 5.1.2 Euclidean MaxMatching:** *Assume that $n$ is even. A feasible solution $s$ is a partition of $\{1, \ldots, n\}$ into $n/2$ pairs $E_1, \ldots, E_{n/2}$ where $E_i = (a_i, b_i)$. The cost of $s$ on $P = \{p_1, \ldots, p_n\}$ is given by*

$$\mathit{cost}_{\mathit{maxmatching}}^{(n,s)}(p_1, \ldots, p_n) = \sum_{i=1}^{n/2} d(p_{a_i}, p_{b_i}) \ .$$

**Example 5.1.3 Euclidean MaxTSP:** *A feasible solution* $s$ *is a permutation of* $\{1,\dots,n\}$*. The cost of* $s$ *on* $P = \{p_1,\dots,p_n\}$ *is given by*

$$cost^{(n,s)}_{maxtsp}(p_1,\dots,p_n) = d(p_{s(n)},p_{s(1)}) + \sum_{i=1}^{n-1} d(p_{s(i)},p_{s(i+1)}) \ .$$

**Example 5.1.4 Average Distance:** *Since we only want to estimate the value of the average distance, there is only one feasible solution* $s$*. For technical reasons we scale the average distance of the points by* $n$ *and denote the costfunction of the solution by:*

$$cost^{(n,s)}_{avgdistance}(p_1,\dots,p_n) = \frac{1}{n-1}\sum_{i=1}^{n}\sum_{j\in\{1,\dots,n\}\setminus\{i\}} d(p_i,p_j) \ .$$

Since the definition of the solution is oblivious of the position of points we can speak of the change of the cost of solution $s$ when moving from point set $P$ to another point set $P'$. In particular our coreset construction needs the following two conditions.

**Definition 5.1.5 ($\ell$-Lipschitz)** *Let* $\ell \geq 1$ *be a constant. We say that* $cost^{(n,s)}_{\Pi}$ *is* $\ell$-Lipschitz*, if for arbitrary points* $p_1,\dots,p_n$ *and* $p'_1,\dots,p'_n$ *with* $\sum_{i=1}^{n} d(p_i,p'_i) \leq D$ *we have*

$$\left|cost^{(n,s)}_{\Pi}(p_1,\dots,p_n) - cost^{(n,s)}_{\Pi}(p'_1,\dots,p'_n)\right| \leq \ell \cdot D$$

*and*

$$p_1 = p_2 = \dots = p_n \Longrightarrow cost^{(n,s)}_{\Pi}(p_1,\dots,p_n) = 0 \ .$$

*We call* $\Pi$ $\ell$*-Lipschitz, if for every* $n \in \mathbb{N}$ *and* $s \in \mathcal{S}_\Pi(n)$ *the objective function* $cost^{(n,s)}_{\Pi}$ *is* $\ell$*-Lipschitz.*

**Definition 5.1.6 ($\lambda$-mean preserving)** *Let* $\lambda \leq 1$ *be a constant. We say that* $\Pi$ *is* $\lambda$-mean preserving*, if for any point set* $P$ *in* $\mathbb{R}^d$ *we have*

$$Opt_\Pi(P) \geq \lambda \cdot \sum_{p\in P} d(p,\mu) \ ,$$

*where* $\mu := \mu(P)$ *is the mean or center of gravity of* $P$ *(see Section 2.1).*

We will show that all optimization problems stated before are $\ell$-Lipschitz and $\lambda$-means preserving with constants $\ell$ and $\lambda$.

**Lemma 5.1.7** *The Euclidean MaxCut problem is* $\ell$*-Lipschitz with* $\ell = 1$ *and* $\lambda$*-mean preserving with* $\lambda = \frac{1}{4}$*.*

**Proof :** We first show that MaxCut is 1-Lipschitz.

$$\begin{aligned}
& \left|cost_{\Pi}^{(n,s)}(p_1,\ldots,p_n) - cost_{\Pi}^{(n,s)}(p'_1,\ldots,p'_n)\right| \\
= & \left|\frac{1}{n}\sum_{i\in C_1}\sum_{j\in C_2} d(p_i,p_j) - \frac{1}{n}\sum_{i\in C_1}\sum_{j\in C_2} d(p'_i,p'_j)\right| \\
\leq & \frac{1}{n}\sum_{i\in C_1}\sum_{j\in C_2}\left|d(p_i,p_j) - d(p'_i,p'_j)\right| \\
= & \frac{1}{n}\sum_{i\in C_1}\sum_{j\in C_2}\max\{d(p_i,p_j) - d(p'_i,p'_j), d(p'_i,p'_j) - d(p_i,p_j)\} \\
\leq & \frac{1}{n}\sum_{i\in C_1}\sum_{j\in C_2}\max\{d(p_i,p'_i) + d(p'_i,p'_j) + d(p'_j,p_j) - d(p'_i,p'_j), \\
& d(p'_i,p_i) + d(p_i,p_j) + d(p_j,p'_j) - d(p_i,p_j)\} \\
= & \frac{1}{n}\sum_{i\in C_1}\sum_{j\in C_2}(d(p_i,p'_i) + d(p'_j,p_j)) \\
\leq & \frac{1}{n}\sum_{i\in C_1} n\cdot d(p_i,p'_i) + \frac{1}{n}\cdot n\cdot\sum_{j\in C_2} d(p_j,p'_j) \\
= & \sum_{i=1}^{n} d(p_i,p'_i) = D
\end{aligned}$$

To show that MaxCut is $\lambda$-mean preserving we first we show the following known inequality (also shown in [39]):

$$d(p,\mu) \leq \frac{1}{n}\sum_{q\in P} d(p,q) \ . \tag{5.1}$$

First, note that by projecting all points on the line through $p$ and $\mu$, the left side does not change and the right side does not increase. We can therefore assume that all points lie on a line, i.e. that all points are real numbers. For real numbers we obtain:

$$\begin{aligned}
d(p,\mu) &= |p-\mu| = |p - \frac{1}{n}\cdot\sum_{q\in P} q| = \frac{1}{n}\cdot|n\cdot p - \sum_{q\in P} q| \\
&= \frac{1}{n}\cdot|\sum_{q\in P}(p-q)| \leq \frac{1}{n}\sum_{q\in P}|p-q| \ .
\end{aligned}$$

We now show that MaxCut is $\lambda$-mean preserving with $\lambda = \frac{1}{4}$. We first consider a random cut $C_1, C_2$. For each $i \in \{1,...,n\}$ we flip an unbiased coin to decide whether it belongs to $C_1$ or to $C_2$. Since for every pair of indices $i,j$ the probability of separating the indices is $\frac{1}{2}$, the distance from $p_i$ to $p_j$ is counted in the sum with probability $\frac{1}{2}$. The expected value of the objective function $\frac{1}{n}\sum_{i\in C_1}\sum_{j\in C_2} d(p_i,p_j)$ is therefore $\frac{1}{4n}\sum_{p,q\in P} d(p,q)$. Since *Opt* denotes

the maximum value of such a cut, we have

$$Opt \geq \frac{1}{4n} \sum_{p,q \in P} d(p,q)$$

From equation (5.1) it follows that

$$Opt \geq \frac{1}{4} \sum_{p \in P} d(p,\mu)$$

which means that the Euclidean MaxCut problem is $\lambda$-mean preserving with $\lambda = \frac{1}{4}$ □

**Lemma 5.1.8** *The Euclidean Maximum Weighted Matching problem is $\ell$-Lipschitz with $\ell = 1$ and $\lambda$-mean preserving with $\lambda = \frac{1}{2}$.*

**Proof :** We first show that Euclidean Maximum Weighted Matching is 1-Lipschitz.

$$\begin{aligned}
& \left| cost_{\Pi}^{(n,s)}(p_1,\ldots,p_n) - cost_{\Pi}^{(n,s)}(p'_1,\ldots,p'_n) \right| \\
= & \left| \sum_{i=1}^{n/2} d(p_{a_i},p_{b_i}) - \sum_{i=1}^{n/2} d(p'_{a_i},p'_{b_i}) \right| \\
\leq & \sum_{i=1}^{n/2} \left| d(p_{a_i},p_{b_i}) - d(p'_{a_i},p'_{b_i}) \right| \\
= & \sum_{i=1}^{n/2} \max\{d(p_{a_i},p_{b_i}) - d(p'_{a_i},p'_{b_i}), d(p'_{a_i},p'_{b_i}) - d(p_{a_i},p_{b_i})\} \\
\leq & \sum_{i=1}^{n/2} \max\{d(p_{a_i},p'_{a_i}) + d(p'_{a_i},p'_{b_i}) + d(p'_{b_i},p_{b_i}) - d(p'_{a_i},p'_{b_i}), \\
& d(p'_{a_i},p_{a_i}) + d(p_{a_i},p_{b_i}) + d(p_{b_i},p'_{b_i}) - d(p_{a_i},p_{b_i})\} \\
= & \sum_{i=1}^{n/2} (d(p_{a_i},p'_{a_i}) + d(p'_{b_i},p_{b_i})) \\
= & \sum_{i=1}^{n} d(p_i,p'_i) = D
\end{aligned}$$

where the last equality comes from the fact that $\{1,...,n\}$ is partitioned into pairs $(a_i,b_i)$.

To show that it is $\frac{1}{2}$-mean preserving we look at a random matching constructed the following way: We connect the first index $i = 1$ to one other index $j \in \{1,\ldots,n\} \setminus \{1\}$ uniformly chosen from all other indices. Then we delete both indices from $\{1,...,n\}$ and go on with this construction until all indices are matched. Notice that each pair $(i,j)$ with $i \neq j$ belongs to our matching with probability $\frac{1}{n-1}$. When $M$ denotes the aggregated cost of this matching, then

$$Opt \geq \mathbf{E}[M] = \frac{1}{2(n-1)} \sum_{i,j=1}^{n} d(p_i, p_j) \geq \frac{1}{2n} \sum_{i,j=1}^{n} d(p_i, p_j) \geq \frac{1}{2} \sum_{p \in P} d(p, \mu)$$

where the last inequality follows from equation 5.1.

□

**Lemma 5.1.9** *The Euclidean Maximum Travelling Salesman problem is $\ell$-Lipschitz with $\ell = 2$ and $\lambda$-mean preserving with $\lambda = 1$.*

**Proof :** We first show that Euclidean Maximum Travelling Salesman is 2-Lipschitz.

$$\begin{aligned}
& \left| cost_{\Pi}^{(n,s)}(p_1, \ldots, p_n) - cost_{\Pi}^{(n,s)}(p'_1, \ldots, p'_n) \right| \\
= \; & \left| d(p_{s(n)}, p_{s(1)}) + \sum_{i=1}^{n-1} d(p_{s(i)}, p_{s(i+1)}) - d(p'_{s(n)}, p'_{s(1)}) - \sum_{i=1}^{n-1} d(p'_{s(i)}, p'_{s(i+1)}) \right| \\
\leq \; & \left| d(p_{s(n)}, p_{s(1)}) - d(p'_{s(n)}, p'_{s(1)}) \right| + \sum_{i=1}^{n-1} \left| d(p_{s(i)}, p_{s(i+1)}) - d(p'_{s(i)}, p'_{s(i+1)}) \right| \\
= \; & \max\{d(p_{s(n)}, p_{s(1)}) - d(p'_{s(n)}, p'_{s(1)}), d(p'_{s(n)}, p'_{s(1)}) - d(p_{s(n)}, p_{s(1)})\} \\
& + \sum_{i=1}^{n-1} \max\{d(p_{s(i)}, p_{s(i+1)}) - d(p'_{s(i)}, p'_{s(i+1)}), d(p'_{s(i)}, p'_{s(i+1)}) - d(p_{s(i)}, p_{s(i+1)})\} \\
\leq \; & \max\{d(p_{s(n)}, p'_{s(n)}) + d(p'_{s(n)}, p'_{s(1)}) + d(p'_{s(1)}, p_{s(1)}) - d(p'_{s(n)}, p'_{s(1)}), \\
& d(p'_{s(n)}, p_{s(n)}) + d(p_{s(n)}, p_{s(1)}) + d(p_{s(1)}, p'_{s(1)}) - d(p_{s(n)}, p_{s(1)})\} \\
& + \sum_{i=1}^{n-1} \max\{d(p_{s(i)}, p'_{s(i)}) + d(p'_{s(i)}, p'_{s(i+1)}) + d(p'_{s(i+1)}, p_{s(i+1)}) - d(p'_{s(i)}, p'_{s(i+1)}), \\
& d(p'_{s(i)}, p_{s(i)}) + d(p_{s(i)}, p_{s(i+1)}) + d(p_{s(i+1)}, p'_{s(i+1)}) - d(p_{s(i)}, p_{s(i+1)})\} \\
= \; & d(p_{s(n)}, p'_{s(n)}) + d(p'_{s(1)}, p_{s(1)}) + \sum_{i=1}^{n-1} d(p_{s(i)}, p'_{s(i)}) + d(p'_{s(i+1)}, p_{s(i+1)}) \\
= \; & 2 \sum_{i=1}^{n} d(p_{s(i)}, p'_{s(i)}) = 2 \sum_{i=1}^{n} d(p_i, p'_i) = 2 \cdot D \; .
\end{aligned}$$

To show that it is 1-mean preserving we look at a random permutation $s$ chosen uniformly from the set of permutations: Notice that each possible edge $(i, j)$ with $i \neq j$ appears in the sum with probability $\frac{1}{n-1} + \frac{1}{n-2} \geq \frac{2}{n-1}$. When $M$ denotes the aggregated cost of this solution, then

$$Opt \geq \mathbf{E}[M] \geq \frac{1}{n-1} \sum_{i,j=1}^{n} d(p_i, p_j) \geq \frac{1}{n} \sum_{i,j=1}^{n} d(p_i, p_j) \geq \sum_{p \in P} d(p, \mu)$$

where the last inequality follows from equation 5.1. □

**Lemma 5.1.10** *The Euclidean Average Distance problem is $\ell$-Lipschitz with $\ell = 4$ and $\lambda$-mean preserving with $\lambda = 1$.*

**Proof :** We first show that Euclidean Average Distance is 4-Lipschitz.

$$\begin{aligned}
& \left|cost_{\Pi}^{(n,s)}(p_1,\ldots,p_n) - cost_{\Pi}^{(n,s)}(p'_1,\ldots,p'_n)\right| \\
= & \left|\frac{1}{n-1}\sum_{i=1}^{n}\sum_{j\in\{1,\ldots,n\}\setminus\{i\}} d(p_i,p_j) - \frac{1}{n-1}\sum_{i=1}^{n}\sum_{j\in\{1,\ldots,n\}\setminus\{i\}} d(p'_i,p'_j)\right| \\
\leq & \frac{1}{n-1}\sum_{i=1}^{n}\sum_{j\in\{1,\ldots,n\}\setminus\{i\}} \left|d(p_i,p_j) - d(p'_i,p'_j)\right| \\
= & \frac{1}{n-1}\sum_{i=1}^{n}\sum_{j\in\{1,\ldots,n\}\setminus\{i\}} \max\{d(p_i,p_j) - d(p'_i,p'_j), d(p'_i,p'_j) - d(p_i,p_j)\} \\
\leq & \frac{1}{n-1}\sum_{i=1}^{n}\sum_{j\in\{1,\ldots,n\}\setminus\{i\}} \max\{d(p_i,p'_i) + d(p'_i,p'_j) + d(p'_j,p_j) - d(p'_i,p'_j), \\
& d(p'_i,p_i) + d(p_i,p_j) + d(p_j,p'_j) - d(p_i,p_j)\} \\
= & \frac{1}{n-1}\sum_{i=1}^{n}\sum_{j\in\{1,\ldots,n\}\setminus\{i\}} d(p_i,p'_i) + d(p_j,p'_j) \\
\leq & \frac{2}{n}\sum_{i,j=1}^{n} 2\cdot d(p_i,p'_i) = 4\cdot\sum_{i=1}^{n} d(p_i,p'_i) = 4\cdot D
\end{aligned}$$

Euclidean Average Distance is 1-mean preserving:

$$\frac{1}{n-1}\sum_{i=1}^{n}\sum_{j\in\{1,\ldots,n\}\setminus\{i\}} d(p_i,p_j) = \frac{1}{n-1}\sum_{i,j=1}^{n} d(p_i,p_j) \geq \frac{n}{n-1}\sum_{i=1}^{n} d(p_i,\mu) \geq \sum_{p\in P} d(p,\mu)$$

□

## 5.1.2 Coresets

Intuitively, a *coreset* is a small weighted set of points that approximates a large (typically unweighted) point set with respect to an optimization problem. We first give a definition of coresets for k-median and k-means clustering [61].

**Definition 5.1.11 (Coresets I.)** *[61] Let $P$ be a weighted set of $n$ points in the $\mathbb{R}^d$. A weighted point set $P_{core}$ in $\mathbb{R}^d$ is an $\epsilon$-coreset for the k-median problem, if for every set $C$ of $k$ centers $(1-\epsilon)\cdot Median(P,C) \leq Median(P_{core},C) \leq (1+\epsilon)\cdot Median(P,C)$ . In a similar way a weighted point set $P_{core}$ is an $\epsilon$-coreset for the k-means clustering problem, if every set $C$ of $k$ centers $(1-\epsilon)\cdot Means(P,C) \leq Means(P_{core},C) \leq (1+\epsilon)\cdot Means(P,C)$ .*

Now we generalize the definition of coresets to arbitrary oblivious optimization problems. Our definition will be only for unweighted point sets. However, by replacing weighted points by multiple copies we can easily generalize this definition to weighted point sets.

**Definition 5.1.12 (Coresets II.)** *Let $\Pi$ be an oblivious optimization problem. Let $P$ be a set of $n$ points in the $\mathbb{R}^d$. A* weighted *set of points $P_{core}$ in the $\mathbb{R}^d$ with weight function $w : P_{core} \to \mathbb{N}$ is an $\epsilon$-coreset for $\Pi$, if there exists a mapping $\gamma : P \to P_{core}$ that satisfies the following constraints.*

- *For every $q \in P_{core}$ we have $|\gamma^{-1}(q)| = w(q)$.*
- *For every solution $s \in \mathcal{S}_\Pi(n)$ we have*

$$\begin{aligned} cost_\Pi^{(n,s)}(p_1,\ldots,p_n) - \epsilon \cdot Opt_\Pi(P) &\leq cost_\Pi^{(n,s)}(\gamma(p_1),\ldots,\gamma(p_n)) \\ &\leq cost_\Pi^{(n,s)}(p_1,\ldots,p_n) + \epsilon \cdot Opt_\Pi(P) \ . \end{aligned}$$

*For an ordered point set $P := (p_1,\ldots,p_n)$ we define $\gamma(P) = (\gamma(p_1),\ldots,\gamma(p_n))$. Since the oblivious optimization problem does not depend on the order of the points, we can define $Opt_\Pi(P_{core}, w) := Opt_\Pi(\gamma(P))$.*

From Definition 5.1.12 the definition of coresets for the problems Euclidean MaxCut, Euclidean MaxTSP, Euclidean MaxMatching, und average distance follows.

**Lemma 5.1.13** *Let $\Pi$ be an oblivious optimization problem, $P$ be a set of $n$ points in the $\mathbb{R}^d$ and let $P_{core} \subset \mathbb{R}^d$ with weight function $w : P_{core} \to \mathbb{N}$ be an $\epsilon$-coreset for $P$ and $\Pi$. Then*

$$Opt_\Pi(P_{core}, w) \in Opt_\Pi(P) \cdot (1 \pm \epsilon)$$

*and*

$$Opt_\Pi(P) \in Opt_\Pi(P_{core}, w) \cdot (1 \pm 2\epsilon) \ .$$

□

## 5.2 Coresets for $k$-Median

We now give a description of our technique to construct coresets of small size. We always assume that all points lie in a bounding cube of sidelength 1, for example $[0,1]^d$. This can be achieved by scaling the points appropriately. Additionally we assume that the optimal objective function value *Opt* of the respective problem is at least $1/\widetilde{\Delta}$. We only need a weak bound $\widetilde{\Delta}$, since all space and time bounds of our algorithms will depend logarithmically on $\widetilde{\Delta}$.

In this section we describe our coreset construction for the k-median problem. The next chapters will adapt the techniques to construct coresets for k-means and oblivious optimization problems.

### 5.2.1 Construction of the Coreset

We impose $Z$ nested square grids $\mathcal{G}_0, \ldots, \mathcal{G}_{Z-1}$ over the point space for some parameter $Z = \left\lceil \log\left(\frac{4\cdot k\cdot 10^d\cdot n(1+\log n)\cdot d^{(d+1)/2}\cdot \tilde{\Delta}}{\epsilon^{d+1}}\right)\right\rceil + 1$. The side length of the grid cells in grid $\mathcal{G}_i$ is $\frac{1}{2^i}$. Our goal will be to identify for each grid $\mathcal{G}_i$ its *heavy cells*, i.e. cells that contain more than a certain threshold of points. This threshold depends on the side length of the grid cells and grows with its inverse (the smaller the cells, the larger the threshold). We will parametrize the threshold by some small value $\delta < n$, which is specified later.

**Definition 5.2.1 (Heavy Cells)** *We call a cell of grid $\mathcal{G}_i$* heavy*, if it contains at least $\delta \cdot 2^i$ points of $P$. A grid cell that is not heavy is* light*.*

Our process consists of two phases. In phase I we determine the coreset points. In phase II we determine their weight. We begin with a description of phase I. The algorithm starts with the coarsest grid $\mathcal{G}_0$. First, it identifies every heavy cell in $\mathcal{G}_0$. Note that grid $\mathcal{G}_0$ consists of only one cell containing all points. Since $\delta < n$ this cell is a heavy cell. Then the algorithm subdivides every heavy cell $\mathcal{C}$ into $2^d$ equal sized quadratic subcells. These subcells are contained in grid $\mathcal{G}_1$. We call $\mathcal{C}$ the *parent cell* of these subcells. If none of these subcells is heavy we place a coreset point in the center of the cell. Otherwise, the algorithm recurses this process with all heavy subcells. The recursion eventually stops because at some point a heavy cell is required to have more than $n$ points inside.

It remains to determine the weight of each coreset point. This is done in phase II of the algorithm. We can think of phase two as 'moving the points' to their corresponding coreset points. The weight of a coreset point is simply the number of points moved to its position. The movement must satisfy the following invariant

(a) every point stays in the smallest heavy cell it is contained it.

By our construction, every heavy cell must contain a coreset point. Thus it is easy to satisfy our invariant. We can simply move every point $p$ of $P$ to an arbitrary coreset point that is contained in the smallest heavy cell containing $p$. Finally, the weights of each coreset point is determined by the number of points moved to it. We will prove that for suitably chosen $\delta$ the resulting weighted set of points $P_{core}$ is an $\epsilon$-coreset for the k-median problem of size $O(k \cdot \log n/\epsilon^d)$.

Below we describe our algorithm in pseudocode. It computes the coreset $P_{core}$ together with weights $w(p)$ for each point $p \in P_{core}$.

```
ComputeCoreset(P, Opt)
    Let H_1, ..., H_h denote the heavy cells in grid G_0
    return ⋃_{i=1}^{h} ComputeCoresetPoints(H_i)
```

```
COMPUTECORESETPOINTS (cell C)
  if C has no heavy subcells then
    p ← center of C
    w(p) ← number of points in C
    return {p}
  else
    Let H_1, ..., H_h denote the heavy subcells of C
    Let L_1, ..., L_ℓ denote the light subcells of C
    Let m denote the number of points in ⋃_{i=1}^{ℓ} L_i
    P_core = ⋃_{i=1}^{h} COMPUTECORESETPOINTS(H_i)
    Let q be an arbitrary point in P_core
    w(q) ← w(q) + m
    return P_core
```

We will now prove that the point set computed by COMPUTECORESET is indeed an $\epsilon$-coreset for k-median.

We denote by $\mathcal{L}(i)$ the set of non-empty light cells of grid $\mathcal{G}_i$ whose parent cell is heavy. Notice that $\bigcup_i \mathcal{L}(i)$ partitions the plane.

**Claim 5.2.2** *Any point $p \in \mathcal{L}(i)$ is moved a distance of at most $\frac{\sqrt{d}}{2^{i-1}}$ during our coreset construction.*

**Proof :** Our invariant assures that every point $p$ stays within the smallest heavy cell it is contained in. Every point $p$ that is contained in $\mathcal{L}(i)$ is contained in a heavy cell in grid $\mathcal{G}_{i-1}$. Therefore, it is moved at most the diagonal length of cells in $\mathcal{G}_{i-1}$, i.e. $\frac{\sqrt{d}}{2^{i-1}}$. □

From now on let C be an arbitrary fixed set of k centers. We partition the sets $\mathcal{L}(i)$ into two subsets $\mathcal{L}_{near}(i)$ and $\mathcal{L}_{dist}(i)$. $\mathcal{L}_{near}(i)$ contains all cells $\mathcal{C}$ whose distance $\min_{q \in C} d(q, \mathcal{C})$ to the nearest center from C is at most $\frac{4}{\epsilon} \cdot \frac{\sqrt{d}}{2^i}$ , i.e.

$$\mathcal{L}_{near}(i) = \{\mathcal{C} \in \mathcal{L}(i) \mid \min_{q \in C} d(q, \mathcal{C}) \leq \frac{4}{\epsilon} \cdot \frac{\sqrt{d}}{2^i}\} \ .$$

$\mathcal{L}_{dist}(i)$ contains all other cells from $\mathcal{L}(i)$, i.e.

$$\mathcal{L}_{dist}(i) = \{\mathcal{C} \in \mathcal{L}(i) \mid \min_{q \in C} d(q, \mathcal{C}) > \frac{4}{\epsilon} \cdot \frac{\sqrt{d}}{2^i}\} \ .$$

**Claim 5.2.3** *The total movement $\sum_i \sum_{p \in \mathcal{L}_{dist}(i)} \frac{\sqrt{d}}{2^{i-1}}$ of points in distant cells satisfies*

$$\sum_i \sum_{p \in \mathcal{L}_{dist}(i)} \frac{\sqrt{d}}{2^{i-1}} \leq \frac{\epsilon}{2} Median(P, C) \ .$$

**Proof :** We use a charging argument from [61].

$$\begin{aligned}\sum_{i}\sum_{p\in\mathcal{L}_{dist}(i)}\frac{\sqrt{d}}{2^{i-1}} &= \frac{\epsilon}{2}\sum_{i}\sum_{p\in\mathcal{L}_{dist}(i)}\frac{4}{\epsilon}\cdot\frac{\sqrt{d}}{2^{i}}\\ &\leq \frac{\epsilon}{2}\cdot \mathit{Median}(P,C)\ ,\end{aligned}$$

where the last inequality follows from the fact that every point in $L_{dist}(i)$ contributes more than $\frac{4}{\epsilon}\cdot\frac{\sqrt{d}}{2^{i}}$ to the cost of the solution C. □

**Claim 5.2.4** *For* $\delta \leq \frac{\epsilon^{d+1}\cdot Opt}{4\cdot k\cdot 10^{d}\cdot(1+\log n)\cdot d^{(d+1)/2}}$ *we get*

$$\sum_{i}\sum_{p\in\mathcal{L}_{near}(i)}\frac{\sqrt{d}}{2^{i-1}} \leq \frac{\epsilon}{2}\mathit{Median}(P,C)\ .$$

**Proof :** We observe that the furthest point in a cell in $\mathcal{L}_{near}(i)$ can have a distance of at most $\frac{\sqrt{d}}{2^{i}}+\frac{4}{\epsilon}\cdot\frac{\sqrt{d}}{2^{i}} = \left(1+\frac{4}{\epsilon}\right)\cdot\frac{\sqrt{d}}{2^{i}}$ to the nearest center. Hence, every cell in $\mathcal{L}_{near}(i)$ is contained in a cube of sidelength $2\cdot\left(1+\frac{4}{\epsilon}\right)\cdot\frac{\sqrt{d}}{2^{i}}$ that is centered at one of the k centers of C. Each of these cubes has volume $\left(2\cdot\left(1+\frac{4}{\epsilon}\right)\cdot\frac{\sqrt{d}}{2^{i}}\right)^{d} \leq \left(\frac{10}{\epsilon}\right)^{d}\cdot\left(\frac{\sqrt{d}}{2^{i}}\right)^{d}$. Every cell in grid $\mathcal{G}_i$ has volume $\left(\frac{1}{2^{i}}\right)^{d}$. Hence, there can be at most $k\cdot\left(\frac{10}{\epsilon}\right)^{d}\cdot d^{d/2}$ cells in $\mathcal{L}_{near}(i)$.

Each of the considered cells is light and so it contains at most $\delta\cdot 2^{i}$ points. Hence for our choice of $\delta$:

$$\begin{aligned}\sum_{i}\sum_{p\in\mathcal{L}_{near}(i)}\frac{\sqrt{d}}{2^{i-1}} &= \sum_{i:\mathcal{L}_{near}(i)\neq\emptyset}\ \sum_{p\in\mathcal{L}_{near}(i)}\frac{\sqrt{d}}{2^{i-1}}\\ &\leq \sum_{i:\mathcal{L}_{near}(i)\neq\emptyset}\left(k\cdot\left(\frac{10}{\epsilon}\right)^{d}\cdot d^{d/2}\right)\cdot\left(\delta\cdot 2^{i}\right)\cdot\left(\frac{\sqrt{d}}{2^{i-1}}\right)\\ &\leq \sum_{i:\mathcal{L}_{near}(i)\neq\emptyset}\delta\cdot\left(2\cdot k\cdot\left(\frac{10}{\epsilon}\right)^{d}\cdot d^{d/2}\right)\cdot\sqrt{d}\\ &\leq \frac{\epsilon}{2}\cdot\left(\frac{1}{1+\log n}\sum_{i:\mathcal{L}_{near}(i)\neq\emptyset}Opt\right)\\ &\leq \frac{\epsilon}{2}\cdot\frac{\mathit{Median}(P,C)}{1+\log n}\cdot|\{i:\mathcal{L}_{near}(i)\neq\emptyset\}|\end{aligned}$$

Now we observe that $\mathcal{L}_{near}(i)\neq\emptyset$ implies that there are non-empty light cells in grid $\mathcal{G}_i$ and heavy cells in grid $\mathcal{G}_{i-1}$. We can have non-empty light cells only if $\delta\cdot 2^{i}>1$, since otherwise

all non-empty cells are heavy. We can have heavy cells in grid $\mathcal{G}_{i-1}$ only if $\delta \cdot 2^{i-1} \leq n$, since otherwise all cells are light. Hence we sum up only over those values of $i$ that satisfy $1/2 < \delta \cdot 2^{i-1} \leq n$. Clearly, these are at most $1 + \log n$ distinct values and so we get

$$\frac{\epsilon}{2} \cdot \frac{\mathit{Median}(P, C)}{1 + \log n} \cdot |\{i : \mathcal{L}_{\mathit{near}}(i) \neq \emptyset\}| \leq \frac{\epsilon}{2} \cdot \mathit{Median}(P, C) \ ,$$

which concludes the proof of Claim 5.2.4 □

**Lemma 5.2.5** *The set $P_{\mathit{core}}$ is an $\epsilon$-coreset for $\delta \leq \frac{\epsilon^{d+1} \cdot \mathit{Opt}}{4 \cdot k \cdot 10^d \cdot (1+\log n) \cdot d^{(d+1)/2}}$.*

**Proof :** We first observe that every point of $P$ is contained in some cell in $\bigcup_i \mathcal{L}(i)$. By Claim 5.2.2 we know that every point that is contained in a grid cell in $\mathcal{L}(i)$ is moved a distance of at most $\frac{\sqrt{d}}{2^{i-1}}$. Therefore, Claims 5.2.3 and 5.2.4 imply that the points are moved an overall distance of at most $\epsilon \mathit{Median}(P, C)$. Finally, we observe that the cost of any set of $k$ centers changes by at most $\pm D$ when the points of the point set $P$ are moved by an overall distance of $D$. Hence the set $P_{\mathit{core}}$ constructed by our algorithm is an $\epsilon$-coreset for $k$-median. □

## 5.2.2 Size of the Coreset

Our next step is to give an upper bound on the size of the coreset. For every grid $\mathcal{G}_i$ we define $\mathcal{M}(i)$ to be the set of heavy cells that do not have a heavy subcell. Notice that the cells $\mathcal{M}(i)$ are exactly those cells that contain a coreset point. Hence, it will be sufficient to determine the cardinality of this set.

**Definition 5.2.6 (Center cells)** *Let $C_{\mathit{opt}}$ denote an optimal set of $k$ centers for the $k$-median problem. A cell $\mathcal{C}$ in grid $\mathcal{G}_i$ is called a* center cell*, if its distance to the nearest center in $C_{\mathit{opt}}$ is less than $\frac{\sqrt{d}}{2^{i+1}}$, i.e. if $\min_{q \in C_{\mathit{opt}}} d(q, \mathcal{C}) < \frac{\sqrt{d}}{2^{i+1}}$.*

*We define $\mathcal{M}_{\mathit{center}}(i) = \{\mathcal{C} \in \mathcal{M}(i) | \mathcal{C} \text{ is center cell}\}$ to be the subset of $\mathcal{M}(i)$ that are center cells. We use $\mathcal{M}_{\mathit{external}}(i) = \mathcal{M}(i) \setminus \mathcal{M}_{\mathit{center}}(i)$ to denote the remaining cells of $\mathcal{M}(i)$. We call these cells* external *cells.*

**Claim 5.2.7** *Every external cell contributes at least $\delta \cdot \sqrt{d}/2$ to the cost Opt of $C_{\mathit{opt}}$.*

**Proof :** Every external cell $\mathcal{C} \in \mathcal{M}(i)$ is a heavy cell and so it contains at least $\delta \cdot 2^i$ points. Each point contributes at least $\frac{\sqrt{d}}{2^{i+1}}$ to the cost of the optimal solution. Hence the overall contribution of the points in $\mathcal{C}$ is at least $\delta \cdot \sqrt{d}/2$. □

**Lemma 5.2.8** *If $\delta \geq \frac{\epsilon^{d+1} \cdot \mathit{Opt}}{8 \cdot k \cdot 10^d \cdot (1+\log n) \cdot d^{(d+1)/2}}$, the size of the coreset is at most $\frac{17 \cdot k \cdot 10^d \cdot (2+\log n) \cdot d^{d/2}}{\epsilon^{d+1}} = O(k \cdot \log n / \epsilon^{d+1})$.*

**Proof :** From Claim 5.2.7 it follows that

$$\left|\bigcup_i \mathcal{M}_{external}(i)\right| \leq \frac{2 \cdot Opt}{\delta \cdot \sqrt{d}} \ .$$

All cells from $\mathcal{M}_{center}(i)$ are contained in $k$ cubes of sidelength $2\left(\frac{\sqrt{d}}{2^i} + \frac{\sqrt{d}}{2^{i+1}}\right) = \frac{3\cdot\sqrt{d}}{2^i}$. Since each cube volume is $\left(\frac{3\cdot\sqrt{d}}{2^i}\right)^d$ and each cell volume is $\left(\frac{\sqrt{d}}{2^i}\right)^d / d^{d/2}$, we know that

$$\left|\mathcal{M}_{center}(i)\right| \leq k \cdot 3^d \cdot d^{d/2} \ .$$

Using similar arguments as in the proof of Claim 5.2.4 we obtain that $\mathcal{M}(i) \neq \emptyset$ for at most $1 + \log n$ distinct values of $i$. Therefore, we get

$$\left|\bigcup_i \mathcal{M}_{center}(i)\right| \leq (\log n + 1) \cdot k \cdot 3^d \cdot d^{d/2} \ .$$

Since $\bigcup_i \mathcal{M}(i) = \bigcup_i \left(\mathcal{M}_{external}(i) \cup \mathcal{M}_{center}(i)\right)$, the size of the coreset is at most $\frac{2\cdot Opt}{\delta\cdot\sqrt{d}} + (\log n + 2) \cdot k \cdot 3^d \cdot d^{d/2}$.

For $\delta \geq \frac{\epsilon^{d+1}\cdot Opt}{8\cdot k\cdot 10^d\cdot(1+\log n)\cdot d^{(d+1)/2}}$, the size of the coreset is at most $\frac{17\cdot k\cdot 10^d\cdot(2+\log n)\cdot d^{d/2}}{\epsilon^{d+1}}$. □

### 5.2.3 Finding a suitable value of $\delta$

The coreset construction so far was dependent on a value of $\delta$, which itself was dependent on the unknown value of *Opt*. We show how to find a suitable value of $\delta$.

A good value for $\delta$ can be found using the statements of Lemma 5.2.5 and 5.2.8. Let be $\delta_0 := \frac{\epsilon^{d+1}\cdot Opt}{4k\cdot 10^d(1+\log n)d^{(d+1)/2}}$.

- All coresets constructed with a value $\delta \leq \delta_0$ are $\epsilon$-coresets according to Lemma 5.2.5.
- All coresets constructed with a value $\delta \geq \delta_0/2$ have a size of at most $\mathcal{S} := \frac{17\cdot k\cdot 10^d\cdot(2+\log n)\cdot d^{d/2}}{\epsilon^{d+1}}$ according to Lemma 5.2.8.

For each value of $j \in \{0, 1, \ldots, \lfloor \log(n \cdot \widetilde{\Delta} \cdot \sqrt{d})\rfloor\}$ let be

$$\delta(j) = \frac{\epsilon^{d+1}}{4k \cdot 10^d(1+\log n)d^{(d+1)/2} \cdot \widetilde{\Delta}} \cdot 2^j \ .$$

Denote that the coreset constructed for the highest value of $\delta(j)$ is of size at most $\mathcal{S}$ because this value of $\delta$ is greater than $\delta_0 \cdot n \cdot \sqrt{d}/(2 \cdot Opt)$, which is bigger than $\delta_0/2$, since all points lie in a unit cube.

We want to identify a value $j_0 \in \{0, 1, \ldots, \lfloor \log(n \cdot \widetilde{\Delta} \cdot \sqrt{d})\rfloor\}$, such that

the coreset constructed with the value $\delta = \delta(j_0)$ is of size at most $\mathcal{S}$ and

the coreset constructed with the value $\delta = \delta(j_0 - 1)$ is of size greater than $\mathcal{S}$.

If the coreset constructed with the value $\delta = \delta(0)$ is of size at most $\mathcal{S}$, we set $j_0 = 0$. The value $j_0$ can be easily obtained performing a binary search on the values of $j$.

**Lemma 5.2.9** *The coreset constructed with $\delta = \delta(j_0)$ is an $\epsilon$-coreset for $P$. The size of the coreset is at most $\mathcal{S}$.*

**Proof :** By the choice of $j_0$ the size of the computed coreset is at most $\mathcal{S}$.

We show that the constructed coreset is an $\epsilon$-coreset. If $j_0 = 0$ then the coreset is an $\epsilon$-coreset because $\delta(j_0) = \delta_0/(Opt \cdot \widetilde{\Delta}) \leq \delta_0$.

If $j_0 \neq 0$ then the coreset constructed for the value $\delta = \delta(j_0 - 1) = \delta(j_0)/2$ of size greater than $\mathcal{S}$. Therefore we must have $\delta(j_0)/2 < \delta_0/2$. It follows $\delta(j_0) < \delta_0$ and the computed coreset is an $\epsilon$-coreset. □

In each iteration of the binary search we have to decide if the size of the coreset is at most $\mathcal{S}$. This can be done by constructing the coreset and stopping the process if the number of disjoint heavy cells exceeds $\mathcal{S}$. The marking process yielding to the actual coreset can be seen as a quadtree traversal. Each inner node of this quadtree corresponds to a heavy cell. If on one grid the number of heavy cells exceeds $\mathcal{S}$, we stop the process. Since we have at most $Z = O(\log(n \cdot \widetilde{\Delta}/\epsilon))$ grids, the tree traversal can be done in time $O(\mathcal{S} \cdot \log(\widetilde{\Delta} \cdot n/\epsilon)) = O(k \cdot \log n \cdot \log(\widetilde{\Delta} \cdot n/\epsilon)/\epsilon^{d+1})$.

Since we have $O(\log\log(\widetilde{\Delta} \cdot n))$ iterations of the binary search, the coreset can be constructed in time $O(k \cdot \log n \cdot \log(\widetilde{\Delta} \cdot n/\epsilon) \log\log(\widetilde{\Delta} \cdot n)/\epsilon^{d+1})$.

Note that given a point set $P$ we can first build a quadtree of depth $Z$ of the points in time $O(n \cdot Z) = O(n \cdot \log(n \cdot \widetilde{\Delta}/\epsilon))$. Using this quadtree we can then answer queries on the number of points in certain grid cells in constant time.

**Theorem 11** *Assume that we are given a point set $P \subset [0,1]^d$ of size $n \in \mathbb{N}$ and the guarantee that the value Opt of an optimal $k$-median solution is at least $1/\widetilde{\Delta}$. For each value of $i \in \{0, 1, \ldots, Z\}$ with $Z = \left\lceil \log\left(\frac{4 \cdot k \cdot 10^d \cdot n(1+\log n) \cdot d^{(d+1)/2} \cdot \widetilde{\Delta}}{\epsilon^{d+1}}\right)\right\rceil + 1$ we have a square grid $\mathcal{G}_i$ over the point space, all cells of side length $\frac{1}{2^i}$.*

*Given an oracle, which answers queries on the number of points in grid cells in constant time, we can compute in time $O(k \cdot \log n \cdot \log(\widetilde{\Delta} \cdot n/\epsilon) \cdot \log\log(\widetilde{\Delta} \cdot n)/\epsilon^{d+1})$ an $\epsilon$-coreset of size $O(k \cdot \log n/\epsilon^{d+1})$ for $k$-median.*

*A respective oracle can be constructed in time $O(n \cdot \log(n \cdot \widetilde{\Delta}/\epsilon))$.*

## 5.3 Coresets for $k$-Means

We show how to modify our coreset construction for the k-median problem in a way such that it works for the k-means problem.

### 5.3.1 Construction of the Coreset

We again use $Z$ nestes square grids $\mathcal{G}_0, \ldots, \mathcal{G}_Z$ for $Z = \left\lceil \log\left(\frac{8 \cdot k \cdot 33^{d+1} \cdot n(1+\log n) \cdot d^{(d+2)/2} \cdot \widetilde{\Delta}}{\epsilon^{d+2}}\right)\right\rceil + 1$. The side length of the cells in grid $\mathcal{G}_i$ is $\frac{1}{2^i}$. We use the following modified definition for heavy cells.

**Definition 5.3.1 (Heavy Cells)** *We call a grid cell of grid $\mathcal{G}_i$ heavy for the k-means clustering problem, if it contains at least $\delta \cdot 4^i$ points.*

We use the same invariant as in the construction of the coreset for the k-median problem, i.e. every point stays in the smallest heavy cell it is contained in. Similarly, as in the construction for k-median we denote by $\mathcal{L}(i)$ the set of non-empty light cells in grid $\mathcal{G}_i$ whose parent cell is heavy. We get the following modified version of Claim 5.2.2 with an analogous proof.

**Claim 5.3.2** *Any point $p \in \mathcal{L}(i)$ is moved a distance of at most $\frac{\sqrt{d}}{2^{i-1}}$ during our coreset construction.* □

From now on let C be an arbitrary fixed set of k centers. We partition the sets $\mathcal{L}(i)$ into two subsets $\mathcal{L}_{near}(i)$ and $\mathcal{L}_{dist}(i)$. $\mathcal{L}_{near}(i)$ contains all cells $\mathcal{C}$ whose distance $\min_{q \in C} d(q, \mathcal{C})$ to the nearest center from C is at most $\frac{16}{\epsilon} \cdot \frac{\sqrt{d}}{2^i}$, i.e.

$$\mathcal{L}_{near}(i) = \{\mathcal{C} \in \mathcal{L}(i) \mid \min_{q \in C} d(q, \mathcal{C}) \leq \frac{16}{\epsilon} \cdot \frac{\sqrt{d}}{2^i}\} \ .$$

$\mathcal{L}_{dist}(i)$ contains all other cells from $\mathcal{L}(i)$, i.e.

$$\mathcal{L}_{dist}(i) = \{\mathcal{C} \in \mathcal{L}(i) \mid \min_{q \in C} d(q, \mathcal{C}) > \frac{16}{\epsilon} \cdot \frac{\sqrt{d}}{2^i}\} \ .$$

**Claim 5.3.3** *For any point $p \in P$ let $\hat{p}$ denote its corresponding coreset point. Then we have*

$$\Big|\sum_i \sum_{p \in \mathcal{L}_{dist}(i)} d(p, C)^2 - \sum_i \sum_{p \in \mathcal{L}_{dist}(i)} d(\hat{p}, C)^2\Big| \leq \frac{\epsilon}{2} \mathit{Means}(P, C)$$

**Proof :** We use a charging argument similar to one from [61]. Let $p$ be an arbitrary point in $P$ and $\hat{p}$ be the corresponding coreset point. By Claim 5.3.2 we know that $d(p, \hat{p}) \leq \frac{\sqrt{d}}{2^{i-1}}$. We get

$$d(\hat{p}, C)^2 \leq \left(d(p, C) + \frac{\sqrt{d}}{2^{i-1}}\right)^2 = d(p, C)^2 + 2d(p, C) \cdot \frac{\sqrt{d}}{2^{i-1}} + \frac{d}{4^{i-1}}$$

and since $d(p, C) > \frac{\sqrt{d}}{2^{i-1}}$:

$$d(\hat{p}, C)^2 \geq \left(d(p, C) - \frac{\sqrt{d}}{2^{i-1}}\right)^2 \geq d(p, C)^2 - 2d(p, C) \cdot \frac{\sqrt{d}}{2^{i-1}} - \frac{d}{4^{i-1}} \ .$$

We also know that $d(p, C) \geq \frac{16}{\epsilon} \cdot \frac{\sqrt{d}}{2^i}$ and so

$$\begin{aligned} \left|d(p, C)^2 - d(\hat{p}, C)^2\right| &\leq 2d(p, C) \cdot \frac{\sqrt{d}}{2^{i-1}} + \frac{d}{4^{i-1}} \\ &\leq \frac{\epsilon}{4} d(p, C)^2 + \left(\frac{\epsilon}{8}\right)^2 d(p, C)^2 \\ &\leq \frac{\epsilon}{2} d(p, C)^2 \ . \end{aligned}$$

We get

$$\left|\sum_{i}\sum_{p\in\mathcal{L}_{dist(i)}} d(p,C)^2 - \sum_{i}\sum_{p\in\mathcal{L}_{dist(i)}} d(\hat{p},C)^2\right| \leq \sum_{i}\sum_{p\in\mathcal{L}_{dist(i)}} \left|d(p,C)^2 - d(\hat{p},C)^2\right|$$

$$\leq \sum_{i}\sum_{p\in\mathcal{L}_{dist(i)}} \frac{\epsilon}{2} d(p,C)^2 \leq \frac{\epsilon}{2} \mathit{Means}(P,C) \ .$$

□

Our next step is to give an upper bound on the change of the contribution of points in near cells.

**Claim 5.3.4** *For* $\delta \leq \frac{\epsilon^{d+2}\cdot Opt}{8(1+\log n)\cdot k\cdot 33^{d+1}\cdot d^{(d+2)/2}}$ *we get*

$$\Big|\sum_{i:\mathcal{L}_{near}(i)\neq\emptyset}\ \sum_{p\in\mathcal{L}_{near}(i)} \left(d(p,C)^2 - d(\hat{p},C)^2\right)\Big| \leq \frac{\epsilon}{2}\cdot \mathit{Means}(P,C) \ .$$

**Proof :** We use a similar volume argument as in the proof of Claim 5.2.4. Any cell in $\mathcal{L}_{near}(i)$ must be contained in one of $k$ cubes of volume $\left[2\left(\frac{16\sqrt{d}}{\epsilon\cdot 2^i}+\frac{\sqrt{d}}{2^i}\right)\right]^d = \left[\left(\frac{32}{\epsilon}+1\right)\frac{\sqrt{d}}{2^i}\right]^d \leq \left(\frac{33}{\epsilon}\right)^d \cdot \left(\frac{\sqrt{d}}{2^i}\right)^d$. Each cell has a volume of $\left(\frac{\sqrt{d}}{\sqrt{d}\cdot 2^i}\right)^d$. Therefore, there can be at most $k\cdot\left(\frac{33}{\epsilon}\right)^d\cdot d^{d/2}$ such cells.

Let $p$ be an arbitrary point in $\mathcal{L}_{near}(i)$ and $\hat{p}$ be the corresponding coreset point. We will show $d(\hat{p},C)^2 - d(p,C)^2 \leq \left(3+\frac{16}{\epsilon}\right)\cdot\frac{d}{4^{i-1}}$ and $d(\hat{p},C)^2 - d(p,C)^2 \geq -\left(3+\frac{16}{\epsilon}\right)\cdot\frac{d}{4^{i-1}}$:

We observe that $d(p,C) \leq (1+\frac{8}{\epsilon})\cdot\frac{\sqrt{d}}{2^{i-1}}$. We get

$$\begin{aligned} d(\hat{p},C)^2 &\leq \left(d(p,C)+\frac{\sqrt{d}}{2^{i-1}}\right)^2 \\ &\leq d(p,C)^2 + 2d(p,C)\cdot\frac{\sqrt{d}}{2^{i-1}} + \frac{d}{4^{i-1}} \\ &\leq d(p,C)^2 + 2\left(1+\frac{8}{\epsilon}\right)\cdot\frac{\sqrt{d}}{2^{i-1}}\cdot\frac{\sqrt{d}}{2^{i-1}} + \frac{d}{4^{i-1}} \\ &\leq d(p,C)^2 + \left(3+\frac{16}{\epsilon}\right)\cdot\frac{d}{4^{i-1}} \end{aligned}$$

*Case A:* $d(p,C) \geq \frac{\sqrt{d}}{2^{i-1}}$. Then:

$$\begin{aligned} d(\hat{p},C)^2 &\geq \left(d(p,C) - \frac{\sqrt{d}}{2^{i-1}}\right)^2 \\ &\geq d(p,C)^2 - 2d(p,C)\cdot\frac{\sqrt{d}}{2^{i-1}} - \frac{d}{4^{i-1}} \\ &\geq d(p,C)^2 - 2\left(1+\frac{8}{\epsilon}\right)\cdot\frac{\sqrt{d}}{2^{i-1}}\cdot\frac{\sqrt{d}}{2^{i-1}} - \frac{d}{4^{i-1}} \\ &\geq d(p,C)^2 - \left(3+\frac{16}{\epsilon}\right)\cdot\frac{d}{4^{i-1}} \end{aligned}$$

*Case B:* $d(p,C) < \frac{\sqrt{d}}{2^{i-1}}$. Then

$$d(\hat{p},C)^2 \geq 0 \geq d(p,C)^2 - \frac{d}{4^{i-1}} \geq d(p,C)^2 - \left(3+\frac{16}{\epsilon}\right)\cdot\frac{d}{4^{i-1}}$$

Altogether we obtain $\left|d(\hat{p},C)^2 - d(p,C)^2\right| \leq \left(3+\frac{16}{\epsilon}\right)\cdot\frac{d}{4^{i-1}}$.

Each of the considered cells is light and so it contains at most $\delta\cdot 4^i$ points. Hence for our choice of $\delta$:

$$\begin{aligned} &\left|\sum_{i:\mathcal{L}_{near}(i)\neq\emptyset}\sum_{p\in\mathcal{L}_{near}(i)}\left(d(p,C)^2 - d(\hat{p},C)^2\right)\right| \\ \leq\ &\sum_{i:\mathcal{L}_{near}(i)\neq\emptyset}\sum_{p\in\mathcal{L}_{near}(i)}\left|d(p,C)^2 - d(\hat{p},C)^2\right| \\ \leq\ &\sum_{i:\mathcal{L}_{near}(i)\neq\emptyset}\underbrace{\delta\cdot 4^i}_{\text{number of points per cell}}\cdot\underbrace{k\cdot\left(\frac{33}{\epsilon}\right)^d\cdot d^{d/2}}_{\text{number of cells}}\cdot\left(3+\frac{16}{\epsilon}\right)\cdot\frac{d}{4^{i-1}} \\ \leq\ &\sum_{i:\mathcal{L}_{near}(i)\neq\emptyset}\delta\cdot 4\cdot k\cdot\left(\frac{33}{\epsilon}\right)^{d+1}\cdot d^{d/2}d \\ \leq\ &\frac{\epsilon}{2}\cdot\left(\frac{1}{1+\log n}\cdot\sum_{i:\mathcal{L}_{near}(i)\neq\emptyset}Opt\right) \\ \leq\ &\frac{\epsilon}{2}\cdot Means(P,C)\ , \end{aligned}$$

where the last inequality follows from a similar argument as in the proof of Claim 5.2.4. □

It follows immediately, that

**Lemma 5.3.5** *The set $P_{core}$ is an $\epsilon$-coreset for $\delta \leq \frac{\epsilon^{d+2}\cdot Opt}{8(1+\log n)\cdot k\cdot 33^{d+1}\cdot d^{(d+2)/2}}$.* □

### 5.3.2 Size of the Coreset

We adapt the proof for the k-median problem to k-means. In the following we give the slightly changed definitions we require. For every grid $\mathcal{G}_i$ we define $\mathcal{M}(i)$ to be the set of heavy cells that do not have a heavy subcell.

**Definition 5.3.6 (Center cells)** *Let $C_{opt}$ denote an optimal set of $k$ centers for the $k$-means problem. A cell $\mathcal{C}$ in grid $\mathcal{G}_i$ is called a* center cell*, if its distance to the nearest center in $C_{opt}$ is less than $\frac{\sqrt{d}}{2^{i+1}}$, i.e. if $\min_{q \in C_{opt}} d(q, \mathcal{C}) < \frac{\sqrt{d}}{2^{i+1}}$.*

We define $\mathcal{M}_{center}(i) = \{\mathcal{C} \in \mathcal{M}(i) | \mathcal{C} \text{ is center cell}\}$ to be the subset of $\mathcal{M}(i)$ that are center cells. We use $\mathcal{M}_{external}(i) = \mathcal{M}(i) \setminus \mathcal{M}_{center}(i)$ to denote the remaining cells of $\mathcal{M}(i)$. We call these cells *external* cells.

**Claim 5.3.7** *Every external cell contributes at least $\delta \cdot d/4$ to the cost Opt of $C_{opt}$.*

**Proof :** Every external cell $\mathcal{C}$ is a heavy cell and so it contains at least $\delta \cdot 4^i$ points. Each point contributes at least $\left(\frac{\sqrt{d}}{2^{i+1}}\right)^2$ to the cost of the optimal solution. Hence the overall contribution of the points in $\mathcal{C}$ is at least $\delta \cdot d/4$. □

**Lemma 5.3.8** *If $\delta \geq \frac{\epsilon^{d+2} \cdot Opt}{16(1+\log n) \cdot k \cdot 33^{d+1} \cdot d^{(d+2)/2}}$, the size of the coreset is at most $\frac{65(1+\log n) \cdot k \cdot 33^{d+1} \cdot d^{d/2}}{\epsilon^{d+2}} = O(k \log n / \epsilon^{d+2})$.*

**Proof :** From Claim 5.3.7 it follows that

$$\left| \bigcup_i \mathcal{M}_{external}(i) \right| \leq \frac{4 \cdot Opt}{\delta \cdot d} \ .$$

All cells from $\mathcal{M}_{center}(i)$ are contained in $k$ cubes of sidelength $2\left(\frac{\sqrt{d}}{2^i} + \frac{\sqrt{d}}{2^{i+1}}\right) = \frac{3 \cdot \sqrt{d}}{2^i}$. Since each cube volume is $\left(\frac{3 \cdot \sqrt{d}}{2^i}\right)^d$ and each cell volume is $\left(\frac{\sqrt{d}}{2^i}\right)^d / d^{d/2}$, we know that

$$\left| \mathcal{M}_{center}(i) \right| \leq k \cdot 3^d \cdot d^{d/2} \ .$$

Using similar arguments as in the proof of Claim 5.2.4 we obtain that $\mathcal{M}(i) \neq \emptyset$ for at most $1 + \log n$ distinct values of $i$. Therefore, we get

$$\left| \bigcup_i \mathcal{M}_{center}(i) \right| \leq (\log n + 1) \cdot k \cdot 3^d \cdot d^{d/2} \ .$$

Since $\bigcup_i \mathcal{M}(i) = \bigcup_i \left(\mathcal{M}_{external}(i) \cup \mathcal{M}_{center}(i)\right)$, the size of the coreset is at most $\frac{4 \cdot Opt}{\delta \cdot d} + (\log n + 1) \cdot k \cdot 3^d \cdot d^{d/2}$. For $\delta \geq \frac{\epsilon^{d+2} \cdot Opt}{16(1+\log n) \cdot k \cdot 33^{d+1} \cdot d^{(d+2)/2}}$, the size of the coreset is at most $\frac{65(1+\log n) \cdot k \cdot 33^{d+1} \cdot d^{d/2}}{\epsilon^{d+2}}$. □

### 5.3.3 Finding a suitable value of $\delta$

To find a suitable value of $\delta$ we use the method of Section 5.2.3. We plug in the values $\delta_0 := \frac{\epsilon^{d+2} \cdot Opt}{8(1+\log n)\cdot k\cdot 33^{d+1}\cdot d^{(d+2)/2}}$ and $\mathcal{S} := \frac{65(1+\log n)\cdot k\cdot 33^{d+1}\cdot d^{d/2}}{\epsilon^{d+2}}$, and set for $j \in \{0,1,\ldots,\lfloor \log(n\cdot\widetilde{\Delta}\cdot\sqrt{d})\rfloor\}$:

$$\delta = \delta(j) = \frac{\epsilon^{d+2}}{8(1+\log n)\cdot k\cdot 33^{d+1}\cdot d^{(d+2)/2}\cdot\widetilde{\Delta}}\cdot 2^j \ .$$

Doing a binary search on the values of $j$ we again find a value $j_0$ such that the coreset constructed with the value $\delta = \delta(j_0)$ is of size at most $\mathcal{S}$ and the coreset constructed with the value $\delta = \delta(j_0-1)$ is of size greater than $\mathcal{S}$ (if there is no such value we set $j_0 = 0$). Lemma 5.2.9 and it's proof can be stated in the same way and the coreset constructed for this value of $\delta$ is then an $\epsilon$-coreset for k-means.

**Theorem 12** *Assume that we are given a point set $P \subset [0,1]^d$ of size $n \in \mathbb{N}$ and the guarantee that the value Opt of an optimal k-means solution is at least $1/\widetilde{\Delta}$. For each value of $i \in \{0,1,\ldots,Z\}$ with $Z = \left\lceil \log\left(\frac{8\cdot k\cdot 33^{d+1}\cdot n(1+\log n)\cdot d^{(d+2)/2}\cdot\widetilde{\Delta}}{\epsilon^{d+2}}\right)\right\rceil + 1$ we have a square grid $\mathcal{G}_i$ over the point space, all cells of side length $\frac{1}{2^i}$.*

*Given an oracle, which answers queries on the number of points in grid cells in constant time, we can compute in time $O(k\cdot\log n\cdot\log(\widetilde{\Delta}\cdot n/\epsilon)\cdot\log\log(\widetilde{\Delta}\cdot n)/\epsilon^{d+2})$ an $\epsilon$-coreset of size $O(k\cdot\log n/\epsilon^{d+2})$ for k-means.*

*A respective oracle can be constructed in time $O(n\cdot\log(n\cdot\widetilde{\Delta}/\epsilon))$.*

## 5.4 Coresets for Oblivious Optimization Problems

Let us assume that $\Pi$ is $\ell$-Lipschitz and $\lambda$-mean preserving. We show that under these conditions our k-Median algorithm run on instance $P$ constructs a weighted point set $P_{core}$ that is an $\epsilon$-coreset for $\Pi$. We modify our proofs of the k-median coreset.

### 5.4.1 Construction of the Coreset

We also use $Z$ nested grids $\mathcal{G}_i$ with side length $\frac{1}{2^i}$ for $Z = \left\lceil \log\left(\frac{n(1+\log n)\cdot d^{(d+1)/2}\cdot(10\ell)^{d+1}\cdot\widetilde{\Delta}}{\epsilon^{d+1}\cdot\lambda^d}\right)\right\rceil + 1$. We use the same definition of heavy cells.

**Definition 5.4.1 (Heavy Cells)** *We call a cell of grid $\mathcal{G}_i$* heavy*, if it contains at least $\delta\cdot 2^i$ points of $P$. A grid cell that is not heavy is* light*.*

We denote by $\mathcal{L}(i)$ the set of light cells of grid $\mathcal{G}_i$ whose parent cell is heavy. Notice that each point is contained in exactly one cell of $\bigcup_i \mathcal{L}(i)$. Claim 5.2.2 still holds.

We partition the sets $\mathcal{L}(i)$ into two subsets $\mathcal{L}_{near}(i)$ and $\mathcal{L}_{dist}(i)$. $\mathcal{L}_{near}(i)$ contains all cells $\mathcal{C}$ whose distance $d(\mu,\mathcal{C})$ to the center of gravity $\mu$ is at most $\frac{4\cdot\ell}{\epsilon\cdot\lambda}\cdot\frac{\sqrt{d}}{2^i}$, i.e.

$$\mathcal{L}_{near}(i) = \{\mathcal{C}\in\mathcal{L}(i) \mid d(\mu,\mathcal{C}) \le \frac{4\cdot\ell}{\epsilon\cdot\lambda}\cdot\frac{\sqrt{d}}{2^i}\} \ .$$

$\mathcal{L}_{dist}(i)$ contains all other cells from $\mathcal{L}(i)$, i.e.

$$\mathcal{L}_{dist}(i) = \{\mathcal{C} \in \mathcal{L}(i) \mid d(\mu, \mathcal{C}) > \frac{4 \cdot \ell}{\epsilon \cdot \lambda} \cdot \frac{\sqrt{d}}{2^i}\} \ .$$

**Claim 5.4.2**

$$\sum_i \sum_{p \in \mathcal{L}_{dist}(i)} \frac{\sqrt{d}}{2^{i-1}} \leq \frac{\epsilon}{2 \cdot \ell} \cdot Opt \ .$$

**Proof :** Any point in $\mathcal{L}_{dist}(i)$ has a distance of more than $\frac{4 \cdot \ell}{\epsilon \cdot \lambda} \cdot \frac{\sqrt{d}}{2^i}$ from the center of gravity $\mu$. Therefore, we get

$$\begin{aligned} \sum_i \sum_{p \in \mathcal{L}_{dist}(i)} \frac{\sqrt{d}}{2^{i-1}} &= \frac{\epsilon \cdot \lambda}{2 \cdot \ell} \sum_i \sum_{p \in \mathcal{L}_{dist}(i)} \frac{4 \cdot \ell}{\epsilon \cdot \lambda} \cdot \frac{\sqrt{d}}{2^i} \\ &\leq \frac{\epsilon \cdot \lambda}{2 \cdot \ell} \sum_i \sum_{p \in \mathcal{L}_{dist}(i)} d(p, \mu) \\ &\leq \frac{\epsilon}{2 \cdot \ell} \cdot Opt \end{aligned}$$

where the last inequality holds because $\Pi$ is $\lambda$-mean preserving. □

**Claim 5.4.3** *For* $\delta \leq \frac{\epsilon^{d+1} \cdot \lambda^d \cdot Opt}{(1+\log n) \cdot d^{(d+1)/2} \cdot (10 \cdot \ell)^{d+1}}$ *we get*

$$\sum_{i:\mathcal{L}(i) \neq \emptyset} \sum_{p \in \mathcal{L}_{near}(i)} \frac{\sqrt{d}}{2^{i-1}} \leq \frac{\epsilon}{2 \cdot \ell} Opt \ .$$

**Proof :** We observe that the furthest point in a cell in $\mathcal{L}_{near}(i)$ can have a distance of at most $(1 + \frac{4\ell}{\epsilon\lambda}) \cdot \frac{\sqrt{d}}{2^i}$ to the center of gravity. Hence, every cell in $\mathcal{L}_{near}(i)$ is contained in a cube of sidelength $2 \cdot \left(1 + \frac{4\ell}{\epsilon\lambda}\right) \frac{\sqrt{d}}{2^i}$. The cube has volume $\left(2 \cdot \left(1 + \frac{4\ell}{\epsilon\lambda}\right) \frac{\sqrt{d}}{2^i}\right)^d \leq \left(\frac{10\ell \cdot \sqrt{d}}{\epsilon\lambda \cdot 2^i}\right)^d$. Since every cell in grid $\mathcal{G}_i$ has volume $\left(\frac{1}{2^i}\right)^d$, there can be at most $\left(\frac{10\ell}{\epsilon\lambda}\right)^d \cdot d^{d/2}$ cells in $\mathcal{L}_{near}(i)$.

Each cell in $\mathcal{L}_{near}(i)$ is light and so it contains at most $\delta \cdot 2^i$ points. Hence,

$$\begin{aligned} &\sum_{i:\mathcal{L}(i) \neq \emptyset} \sum_{p \in \mathcal{L}_{near}(i)} \frac{\sqrt{d}}{2^{i-1}} \\ \leq &\sum_{i:\mathcal{L}(i) \neq \emptyset} \delta \cdot 2^i \cdot \left(\frac{10 \cdot \ell}{\epsilon\lambda}\right)^d \cdot d^{d/2} \cdot \frac{\sqrt{d}}{2^{i-1}} \\ \leq &\frac{\epsilon}{2 \cdot \ell} \cdot Opt \end{aligned}$$

for our choice of $\delta$ and using the same arguments as in Claim 5.2.4. □

**Lemma 5.4.4** *The set $P_{core}$ is an $\epsilon$-coreset for $\delta \leq \frac{\epsilon^{d+1} \cdot \lambda^d \cdot Opt}{(1+\log n) \cdot d^{(d+1)/2} \cdot (10 \cdot \ell)^{d+1}}$.*

**Proof :** We first observe that every point of $P$ is contained in exactly one cell in $\bigcup_i \mathcal{L}(i)$. By Claim 5.2.2 we know that every point that is contained in a grid cell in $\mathcal{L}(i)$ is moved a distance of at most $\frac{\sqrt{d}}{2^{i-1}}$. Therefore, Claims 5.4.2 and 5.4.3 imply that the points are moved an overall distance of at most $\frac{\epsilon}{\ell} Opt$. Since the optimization problem $\Pi$ is $\ell$-Lipschitz we know that the cost of any solution changes by at most $\pm\epsilon \cdot Opt$ under this movement. Hence the set $P_{core}$ constructed by our algorithm is a coreset. □

## 5.4.2 Size of the Coreset

Our next step is to give an upper bound on the size of the coreset. For every grid $\mathcal{G}_i$ we define $\mathcal{M}(i)$ to be the set of heavy cells that do not contain a heavy subcell. Notice that the cells $\mathcal{M}(i)$ are exactly those cells that contain a coreset point. Hence, it will be sufficient to determine the cardinality of this set.

**Definition 5.4.5 (Center Cells)** *Let $\mu$ denote the center of gravity of $P$. A cell $\mathcal{C}$ in grid $\mathcal{G}_i$ is called a* center cell, *if its distance to $\mu$ is at most $\frac{\sqrt{d}}{2^{i+1}}$, i.e. if $d(\mu, \mathcal{C}) \leq \frac{\sqrt{d}}{2^{i+1}}$.*

We define $\mathcal{M}_{center}(i) = \{\mathcal{C} \in \mathcal{M}(i) | \mathcal{C} \text{ is center cell}\}$ to be the subset of $\mathcal{M}(i)$ that are center cells. We use $\mathcal{M}_{external}(i) = \mathcal{M}(i) \setminus \mathcal{M}_{center}(i)$ to denote the remaining cells of $\mathcal{M}(i)$. We call these cells *external* cells. We use $G = \sum_{p \in P} d(p, \mu)$ to denote the sum of distances to the center of gravity.

**Claim 5.4.6** *Every external cell contributes at least $\delta \cdot \sqrt{d}/2$ to $G$.*

**Proof :** Every external cell $\mathcal{C}$ is a heavy cell and so it contains at least $\delta \cdot 2^i$ points. Each point has a distance of at least $\frac{\sqrt{d}}{2^{i+1}}$ to the center of gravity. Hence the overall contribution of the points in $\mathcal{C}$ is at least $\delta \cdot \sqrt{d}/2$. □

**Lemma 5.4.7** *If $\delta \geq \frac{\epsilon^{d+1} \cdot \lambda^d \cdot Opt}{2(1+\log n) \cdot d^{(d+1)/2} \cdot (10 \cdot \ell)^{d+1}}$, the size of the coreset is at most*

$$\frac{4 \cdot (1 + \log n) \cdot d^{d/2} \cdot (10\ell)^{d+1}}{\epsilon^{d+1} \cdot \lambda^{d+1}} = O(\log n / \epsilon^{d+1}) \ .$$

**Proof :** From Claim 5.4.6 and $G = \sum_{p \in P} d(p, \mu) \leq \frac{1}{\lambda} \cdot Opt$ it follows that

$$\Big| \bigcup_i \mathcal{M}_{external}(i) \Big| \leq \frac{2 \cdot Opt}{\delta \cdot \lambda \cdot \sqrt{d}}) \ .$$

All cells from $\mathcal{M}_{center}(i)$ are contained in a cube of sidelength $2\left(\frac{\sqrt{d}}{2^i} + \frac{\sqrt{d}}{2^{i+1}}\right) = \frac{3 \cdot \sqrt{d}}{2^i}$. Since the cube volume is $\left(\frac{3 \cdot \sqrt{d}}{2^i}\right)^d$ and each cell volume is $\left(\frac{1}{2^i}\right)^d$, we know that

$$\big|\mathcal{M}_{center}(i)\big| \leq 3^d \cdot d^{d/2} \ .$$

Using similar arguments as in the proof of Claim 5.2.4 we obtain that $\mathcal{M}(i) \neq \emptyset$ for at most $1 + \log n$ distinct values of $i$. Therefore, we get

$$\Big|\bigcup_i \mathcal{M}_{center}(i)\Big| \leq (\log n + 1) \cdot 3^d \cdot d^{d/2} \ .$$

Since $\bigcup_i \mathcal{M}(i) = \bigcup_i \big(\mathcal{M}_{external}(i) \cup \mathcal{M}_{center}(i)\big)$, the size of the coreset is at most $\frac{2 \cdot Opt}{\delta \cdot \lambda \cdot \sqrt{d}} + (\log n + 2) \cdot 3^d \cdot d^{d/2}$. If $\delta \geq \frac{\epsilon^{d+1} \cdot \lambda^d \cdot Opt}{2(1+\log n) \cdot d^{(d+1)/2} \cdot (10 \cdot \ell)^{d+1}}$, the size of the coreset is at most $\frac{4 \cdot (1+\log n) \cdot d^{d/2} \cdot (10\ell)^{d+1}}{\epsilon^{d+1} \cdot \lambda^{d+1}}$.

□

### 5.4.3 Finding a suitable value of $\delta$

To find a suitable value of $\delta$ we use the method of Section 5.2.3. We plug in the values $\delta_0 := \frac{\epsilon^{d+1} \cdot \lambda^d \cdot Opt}{(1+\log n) \cdot d^{(d+1)/2} \cdot (10 \cdot \ell)^{d+1}}$ and $\mathcal{S} := \frac{4 \cdot (1+\log n) \cdot d^{d/2} \cdot (10\ell)^{d+1}}{\epsilon^{d+1} \cdot \lambda^{d+1}}$, and set for $j \in \{0, 1, \ldots, \lfloor \log(n \cdot \widetilde{\Delta} \cdot \sqrt{d} \cdot \ell) \rfloor\}$:

$$\delta = \delta(j) = \frac{\epsilon^{d+1} \cdot \lambda^d}{(1 + \log n) \cdot d^{(d+1)/2} \cdot (10 \cdot \ell)^{d+1} \cdot \widetilde{\Delta}} \cdot 2^j \ .$$

Doing a binary search on the values of $j$ we again find a value $j_0$ such that the coreset constructed with the value $\delta = \delta(j_0)$ is of size at most $\mathcal{S}$ and the coreset constructed with the value $\delta = \delta(j_0 - 1)$ is of size greater than $\mathcal{S}$ (if there is no such value we set $j_0 = 0$). Lemma 5.2.9 and it's proof can be stated in the same way and the coreset constructed for this value of $\delta$ is then an $\epsilon$-coreset for the respective problem.

**Theorem 13** *Assume that we are given a point set $P \subset [0,1]^d$ of size $n \in \mathbb{N}$ and the guarantee that the value Opt of an optimal solution of the oblivious optimization problem is at least $1/\widetilde{\Delta}$. For each value of $i \in \{0, 1, \ldots, Z\}$ with $Z = \left\lceil \log\left(\frac{n(1+\log n) \cdot d^{(d+1)/2} \cdot (10\ell)^{d+1} \cdot \widetilde{\Delta}}{\epsilon^{d+1} \cdot \lambda^d}\right)\right\rceil + 1$ we have a square grid $\mathcal{G}_i$ over the point space, all cells of side length $\frac{1}{2^i}$.*

*Given an oracle, which answers queries on the number of points in grid cells in constant time, we can compute in time $O(\log n \cdot \log(\widetilde{\Delta} \cdot n/\epsilon) \cdot \log\log(\widetilde{\Delta} \cdot n)/\epsilon^{d+1})$ an $\epsilon$-coreset of size $O(\log n/\epsilon^{d+1})$ for the respective problem.*

*A respective oracle can be constructed in time $O(n \cdot \log(n \cdot \widetilde{\Delta}/\epsilon))$.*

**Corollary 5.4.8** *Given a set $P$ of $n$ points in $\mathbb{R}^d$, a coreset for an oblivious optimization problem can be constructed in time $O(n \cdot \log(n/\epsilon) + \log^2(n/\epsilon) \cdot \log\log n/\epsilon^{d+1})$.*

**Proof :** We first scale the points such that all points lie in $[0,1]^d$ and that there are two points $p, q \in P$ and a dimension $i \in \{1, \ldots, d\}$, such that $|p^{(i)} - q^{(i)}| = 1$. Since the problem is $\lambda$-mean preserving, we can conclude from the triangle inequality that $Opt \geq \lambda/2$. Therefore we can set $\widetilde{\Delta} := 2/\lambda$ and have $Opt \geq 1/\widetilde{\Delta}$. We apply our coreset technique on the scaled point set. After constructing the coreset we rescale the points. □

## 5.5 Constructing Solutions on the Coreset

In this section we will shortly show how to construct solutions to the various problems after computing the coreset.

### 5.5.1 k-Median

We follow an approach of Har-Peled and Mazumdar [61].

**Lemma 5.5.1** *[61] Given a weighted set $P_{core}$ of $|P_{core}|$ points in $\mathbb{R}^d$ with total weight $n$, one can compute a set $\mathcal{D}$ of size $O(k^2\varepsilon^{-2d}\log^2 n)$ such that at least one subset $C \subset \mathcal{D}$ of size $k$ is a $(1\pm\varepsilon)$-approximate solution to the $k$-median problem on $P_{core}$. The running time of this algorithm is $O(|P_{core}|\cdot\log^2 n + k^5\log^9 n + k^2\varepsilon^{-2d}\log^2 n)$.*

As in [61] we can use this candidate set and an algorithm from [87] to compute a solution:

**Lemma 5.5.2** *[87] Given a weighted point set $P_{core}$ of $|P_{core}|$ points in $\mathbb{R}^d$, with total weight $n$, a set $\mathcal{D}$ of size at most $|P_{core}|$ such that at least one subset $C \subset \mathcal{D}$ of size $k$ is a $(1\pm\varepsilon)$-approximate solution to the $k$-median problem on $P_{core}$, and a parameter $\delta > 0$, one can compute a $(1\pm\varepsilon)$-approximate $k$-median clustering of $P_{core}$ using only centers from $\mathcal{D}$. The overall running time is $O(\rho\cdot|P_{core}|\cdot(\log k)(\log n)\log(1/\delta))$, where $\rho = exp[O((1+\log 1/\varepsilon)/\varepsilon)^d - 1]$. The algorithm succeeds with probability $\geq 1-\delta$.*

**Theorem 14** *On the coreset $P_{core}$ we can compute a $(1\pm\varepsilon)$-approximate $k$-median solution in time*

$$O(\rho k^2\cdot\log k\cdot\log^3 n\cdot\log(1/\delta) + k\cdot\log^3 n\cdot\varepsilon^{-d-1} + k^5\log^9 n + k^2\varepsilon^{-2d}\log^2 n)$$

*where $\rho = exp[O((1+\log 1/\varepsilon)/\varepsilon)^d - 1]$.*

### 5.5.2 k-Means

We again follow an approach of Har-Peled and Mazumdar [61] to compute a $(1\pm\varepsilon)$-approximate $k$-means clustering on $P_{core}$. They use the following lemma:

**Lemma 5.5.3** *[99] Given a weighted set $P_{core}$ of $|P_{core}|$ points in $\mathbb{R}^d$ with total weight $n$, one can compute a set $\mathcal{D}$ of size $O(k^2\varepsilon^{-2d}\log n\cdot\log(\frac{1}{\varepsilon}))$ such that at least one subset $C \subset \mathcal{D}$ of size $k$ is a $(1\pm\varepsilon)$-approximate solution to the $k$-means problem on $P_{core}$. The running time of this algorithm is $O(|P_{core}|\cdot\log(|P_{core}|) + |P_{core}|\cdot\varepsilon^{-d}\log\frac{1}{\varepsilon})$.*

As in [61] we can this candidate set $\mathcal{D}$ to compute a solution. We simply enumerate all $k$-tuples in $\mathcal{D}$, and compute the $k$-means clustering value of each candidate center set. This takes $O(|\mathcal{D}|^k\cdot k\cdot|P_{core}|)$ time. The best tuple provides the required approximation.

**Theorem 15** *On the coreset $P_{core}$ we can compute a $(1\pm\varepsilon)$-approximate $k$-means solution in time*

$$\widetilde{O}(k^{2k+2}\cdot\varepsilon^{-2kd-d-2}\cdot\log^{k+1} n)\ .$$

### 5.5.3 MaxCut

In this section we describe how to compute a MaxCut on the computed coreset. We adapt an algorithm from [39] for unweighted metric MaxCut, which reduces metric MaxCut to MaxCut in big dense weighted graphs. In general we could replace each weighted point by a number of unweighted points and run [39] on the unweighted instance (having $n$ nodes). Such a technique has also been used in [72].

To avoid building the graph of size $n$ we construct a new reduction from MaxCut on coresets to MaxCut in *small* dense graphs. We will prove that the reduction can be done in space and time polylogarithmic in $n$.

We assume that our coreset construction always constructs the coreset points exactly in the middle of a heavy cell. Then we can use a property of our coreset, that each point $p \in P_{core}$ has either a big weight or is far away from the next coreset point. Using this property the techniques of this section follow the ideas of [39].

For every coreset point $p \in P_{core}$ let $w(p)$ denote the weight of point $p$. We also assume that $n = \sum_{p \in P_{core}} w(p)$. For each partition $(L, R)$ of $P_{core}$ we write

$$Cut(P_{core}, L, R) = \sum_{p \in L, q \in R} w(p) \cdot w(q) \cdot d(p, q)$$

and for each complete graph $G = (V, E)$ with weight function $\omega : E \to \mathbb{R}_+$ and each partition $(L, R)$ of $V$ we write

$$Cut(P_{core}, L, R) = \sum_{p \in L, q \in R} \omega(p, q) \ .$$

Recall that *Opt* denotes the value of an optimum cut $(L_{Opt}, R_{Opt})$ of the input point set $P$, scaled by $1/n$:

$$Opt = \max_{(L,R) \text{ partition of } P} \frac{1}{n} \cdot \sum_{p \in L, q \in R} d(p, q) \ .$$

We scale the distances such that the weighted average distance between the coreset points is 1, i.e.

$$\sum_{p,q \in P_{core}} w(p) \cdot w(q) \cdot d(p, q) = n^2 \ .$$

In the following we will always assume that this equality holds.

**Lemma 5.5.4**

$$\frac{n}{4} \leq Opt \leq 2 \cdot n$$

**Proof :** Since $P_{core}$ is an $\epsilon$-coreset for $P$, we have:

$$\begin{aligned} Opt &= \max_{(L,R) \text{ partition of } P} \frac{1}{n} \cdot \sum_{p \in L, q \in R} d(p, q) \\ &\leq \frac{1}{1-\epsilon} \cdot \max_{(L,R) \text{ partition of } P_{core}} \frac{1}{n} \cdot \sum_{p \in L, q \in R} w(p) \cdot w(q) \cdot d(p, q) \leq 2 \cdot n \end{aligned}$$

To show that $Opt \geq n/4$ we consider a random cut $(A, B)$, where each point $p \in P_{core}$ is independently put into $A$ with probability $1/2$ and otherwise put into $B$. The probability of each edge $(p, q)$ to be in the cut is then $1/2$. By linearity of expectation the expected value of the cut is $\mathbf{E}[Cut(P_{core}, A, B)] = 1/2 \cdot \sum_{p,q \in P_{core}} w(p) \cdot w(q) \cdot d(p, \mu) = n^2/2$. Therefore the maximum cut on $P_{core}$ must have a value greater than $n^2/2$. Since *Opt* is the value of a maximum cut on $P$ scaled by $1/n$ and since $P_{core}$ is an $\epsilon$-coreset for $P$, we conclude

$$Opt \geq (1 - \epsilon) \cdot n/2 \geq n/4 \ .$$

□

We will use the coreset $P_{core}$ of an instance started with a value of $\delta$ fulfilling

$$\frac{\epsilon^{d+1} \cdot Opt}{80 \cdot (1 + \log n) \cdot d^{d/2} \cdot 40^d} \leq \delta \leq \frac{\epsilon^{d+1} \cdot Opt}{10 \cdot (1 + \log n) \cdot d^{d/2} \cdot 40^d} \ .$$

We can easily find such a value of $\delta$ because of Lemma 5.5.4. We define

$$\beta := \frac{\epsilon^{d+1}}{80 \cdot (1 + \log n) \cdot d^{d/2} \cdot 40^d}$$

Then we have

$$\beta \leq \frac{\delta}{Opt} \ .$$

**Lemma 5.5.5** *For each point $p \in P_{core}$ let $d(p) := \min\{d(p, q) \mid q \in P_{core}\}$ be the distance to the next coreset point. Then we have:*

$$d(p) \cdot w(p) \geq \frac{\beta \cdot n}{8} \ .$$

**Proof :**

Our algorithm constructs the coreset points beginning with the finest grid $\mathcal{G}_{Z-1}$. In a grid $\mathcal{G}_i$ a coreset point $p$ is introduced in the middle of a cell $\mathcal{C}$, iff $\mathcal{C}$ is heavy and has no heavy subcell (only in that case we would already have a coreset point within $\mathcal{C}$). Since the grid $\mathcal{G}_i$ has side length $\frac{1}{2^i}$ we conclude $d(p) \geq \frac{1}{2 \cdot 2^i}$. Since the cell $\mathcal{C}$ is heavy and all points in the cell are mapped to $p$, we have $w(p) \geq \delta \cdot 2^i$ and therefore $d(p) \cdot w(p) \geq \frac{\delta}{2} \geq \frac{\beta \cdot Opt}{2} \geq \frac{\beta \cdot n}{8}$. After introducting the coreset point $p$ no other coreset point can be introduced within distance $\frac{1}{2 \cdot 2^i}$ of $p$, because the coreset construction goes on with larger cells and the distance to the border of the cell containing $p$ is then again at least $\frac{1}{2 \cdot 2^i}$. Our method introduces no new coreset points in cells already containing a coreset point. Since the weight of a coreset point can only increase during the coreset construction, the lemma follows. □

**Definition 5.5.6 (Distance Weight)** *[39]: The distance weight $\omega_p$ of a coreset point $p \in P_{core}$ is defined as $\omega_p := w(p) \cdot \sum_{q \in P_{core}} w(q) d(p, q)$.*

**Definition 5.5.7 (Graph of Clones)** *We define a weighted complete graph* $W = (X, E)$ *where* $X$ *(called the* set of clones*) is a multiset of points. Each point* $p \in P_{core}$ *is cloned to create* $\left\lfloor \frac{8 \cdot \omega_p}{\epsilon \cdot \beta \cdot n^2} \right\rfloor$ *identical points of* $X$*, and the edge between* $p' \in X$*, a clone of* $p \in P_{core}$*, and* $q' \in X$*, a clone of* $q \in P_{core}$*, has weight*

$$e_{p'q'} := \frac{\epsilon^2 \beta^2 n^4 \cdot w(p) \cdot w(q)}{64 \cdot \omega_p \cdot \omega_q} \cdot d(p, q) \ .$$

*Edges between clones* $p', q'$ *of the same point have weight* $e_{p'q'} := 0$*.*

**Lemma 5.5.8** *:We have for each* $p \in P_{core}$*:*

$$\left\lfloor \frac{8 \cdot \omega_p}{\epsilon \cdot \beta \cdot n^2} \right\rfloor \geq (1 - \epsilon) \cdot \frac{8 \cdot \omega_p}{\epsilon \cdot \beta \cdot n^2} \ .$$

**Proof :** Let $p \in P_{core}$ be an arbitrary coreset point. Then Lemma 5.5.5 shows:

$$\omega_p = \sum_{q \in P_{core}} w(p) \cdot w(q) \cdot d(p, q) \geq \sum_{q \in P_{core}} w(q) \cdot \frac{\beta \cdot n}{8} = \frac{\beta \cdot n^2}{8} \ .$$

Therefore:

$$\frac{8\omega_p}{\epsilon \cdot \beta \cdot n^2} \geq \frac{1}{\epsilon}$$

and the lemma follows. □

We now show that the constructed auxiliary graph $W = (X, E)$ is small.

**Lemma 5.5.9**

$$(1 - \epsilon) \cdot \frac{8}{\epsilon\beta} \leq |X| \leq \frac{8}{\epsilon\beta}$$

**Proof :** We have

$$\begin{aligned} |X| &= \sum_{p \in P_{core}} \left\lfloor \frac{8 \cdot \omega_p}{\epsilon \cdot \beta \cdot n^2} \right\rfloor \geq \sum_{p \in P_{core}} (1 - \epsilon) \cdot \frac{8 \cdot \omega_p}{\epsilon \cdot \beta \cdot n^2} \\ &= (1 - \epsilon) \cdot \sum_{p, q \in P_{core}} \frac{8}{\epsilon \cdot \beta \cdot n^2} \cdot w(p) \cdot w(q) \cdot d(p, q) = (1 - \epsilon) \cdot \frac{8}{\epsilon \cdot \beta} \end{aligned}$$

where the last equality comes from the scaling of the point distances and

$$|X| = \sum_{p \in P_{core}} \left\lfloor \frac{8 \cdot \omega_p}{\epsilon \cdot \beta \cdot n^2} \right\rfloor \leq \sum_{p \in P_{core}} \frac{8 \cdot \omega_p}{\epsilon \cdot \beta \cdot n^2} = \frac{8}{\epsilon \cdot \beta} \ .$$

□

**Lemma 5.5.10** *Each cut* $(L, R)$ *of* $P_{core}$ *corresponds to a cut* $(L', R')$ *of* $W$ *having the value*

$$Cut(W, L', R') \geq (1-\epsilon)^2 \cdot Cut(P_{core}, L, R) \ .$$

*Each cut* $(L', R')$ *of* $W$ *can easily be extended to a cut* $(L, R)$ *of* $P_{core}$ *having the value*

$$Cut(P_{core}, L, R) \geq Cut(W, L', R') \ .$$

**Proof :** Consider a cut $(L, R)$ of $P_{core}$, and let $(L', R')$ be the induced cut of $W$ where all clone vertices of a point $p \in L$ belong to $L'$ and all clone vertices of a point $p \in R$ belong to $R'$. We have:

$$\begin{aligned} & Cut(W, L', R') = \sum_{p' \in L', q' \in R'} e_{p'q'} \\ = & \sum_{p \in L, q \in R} \left\lfloor \frac{8 \cdot \omega_p}{\epsilon \cdot \beta \cdot n^2} \right\rfloor \cdot \left\lfloor \frac{8 \cdot \omega_q}{\epsilon \cdot \beta \cdot n^2} \right\rfloor \cdot \frac{\epsilon^2 \beta^2 n^4 \cdot w(p) \cdot w(q) \cdot d(p,q)}{64 \cdot \omega_p \cdot \omega_q} \\ \geq & \sum_{p \in L, q \in R} (1-\epsilon)^2 \cdot w(p) \cdot w(q) \cdot d(p,q) = Cut(P_{core}, L, R) \ . \end{aligned}$$

On the other hand let $(L', R')$ be an arbitrary cut of $W$. We will first alter the set $(L', R')$ in a way such that the value of the cut does not decrease and all clone vertices of a point $p \in P_{core}$ belong to the same partition.

Consider a point $p \in P_{core}$ and let $v$ be one of the clone vertices. We compute the values

$$C_L := \sum_{w \in L'} e_{v,w}$$

and

$$C_R := \sum_{w \in R'} e_{v,w} \ .$$

Notice that the value of the cut increases by $C_L - C_R$ if we move one clone vertex of $p$ from $L'$ to $R'$. If we move one clone vertex of $p$ from $R'$ to $L'$ the value of the cut decreases by $C_L - C_R$.

If $C_L \geq C_R$ we put all clone vertices of $p$ into $R'$, not decreasing the cut. If $C_L < C_R$ we put all clone vertices of $p$ into $L'$, not decreasing the cut. We do this iteratively for all vertices $p \in P_{core}$. After that for each vertex $p \in P_{core}$ all clone vertices belong to the same partition.

We then construct a cut $(L, R)$ of $P_{core}$ in the following way: We put a point $p \in P_{core}$ into partition $L$ if its clone vertices belong to partition $L'$. Otherwise we put $p$ into $R$. The value of the cut is then:

$$\begin{aligned} & Cut(P_{core}, L, R) = \sum_{p \in L, q \in R} w(p) \cdot w(q) \cdot d(p,q) \\ \geq & \sum_{p \in L, q \in R} \left\lfloor \frac{8 \cdot \omega_p}{\epsilon \cdot \beta \cdot n^2} \right\rfloor \cdot \left\lfloor \frac{8 \cdot \omega_q}{\epsilon \cdot \beta \cdot n^2} \right\rfloor \cdot \frac{\epsilon^2 \beta^2 n^4 \cdot w(p) \cdot w(q) \cdot d(p,q)}{64 \cdot \omega_p \cdot \omega_q} \\ = & \sum_{p' \in L', q' \in R'} e_{p'q'} = Cut(W, L', R') \end{aligned}$$

□

Lemma 5.5.10 shows that each $(1 \pm \epsilon)$-approximate solution of MaxCut on $W$ can be extrapolated to a $(1 \pm \epsilon)^3$-approximate solution of MaxCut on $P_{core}$. It remains to show how to compute an approximate MaxCut on $W$. We will use an algorithm of [51] to compute such an approximate solution. The algorithm works for so called *dense graphs*. We can show that $W$ is dense by showing that the maximum weight of an edge is at most a constant factor larger than the average edge weight.

**Lemma 5.5.11** $\max_{p',q' \in X}(e_{p'q'}) \leq 16 \cdot \text{avg}_{p',q' \in X}(e_{p'q'})$

**Proof :** The average value of an edge $e \in E$ is:

$$\begin{aligned} & \frac{\sum_{p,q \in P_{core}} \left\lfloor \frac{8 \cdot \omega_p}{\epsilon \cdot \beta \cdot n^2} \right\rfloor \cdot \left\lfloor \frac{8 \cdot \omega_q}{\epsilon \cdot \beta \cdot n^2} \right\rfloor \frac{\epsilon^2 \beta^2 n^4 \cdot w(p) \cdot w(q) \cdot d(p,q)}{64 \cdot \omega_p \cdot \omega_q}}{|X|^2} \\ \geq & \frac{\sum_{p,q \in P_{core}} (1-\epsilon)^2 \cdot w(p) \cdot w(q) \cdot d(p,q)}{|X|^2} = (1-\epsilon)^2 \frac{n^2}{|X|^2} \geq (1-\epsilon)^2 \frac{n^2 \cdot \epsilon^2 \cdot \beta^2}{64} \end{aligned}$$

where the last inequality comes from Lemma 5.5.9.

To show an upper bound for all edge weights we use that for each point $p \in P_{core}$:

$$\begin{aligned} \frac{\omega_p}{w(p)} &= \sum_{q \in P_{core}} w(q) \cdot d(p,q) = \frac{1}{n} \left( \sum_{q,z \in P_{core}} w(q) \cdot w(z) \cdot d(p,q) \right) \\ &= \frac{1}{2n} \left( \left( \sum_{q,z \in P_{core}} w(q) \cdot w(z) \cdot d(p,q) \right) + \left( \sum_{q,z \in P_{core}} w(q) \cdot w(z) \cdot d(p,z) \right) \right) \\ &\geq \frac{1}{2n} \cdot \sum_{q,z \in P_{core}} w(q) \cdot w(z) \cdot d(q,z) = \frac{n}{2} \end{aligned}$$

Now take an arbitrary edge between clone vertices of $p$ and $q$. Using the triangle inequality and the fact that for arbitrary $a, b \geq 1$ we have $\frac{a+b}{a \cdot b} \leq \frac{2}{\min\{a,b\}}$, we can bound the weight of the edge as follows:

$$\begin{aligned} e_{p'q'} &= \frac{d(p,q) \cdot \epsilon^2 \beta^2 n^4 \cdot w(p) \cdot w(q)}{64 \cdot \omega_p \cdot \omega_q} \\ &\leq \frac{\frac{1}{n} \left( \sum_{z \in P_{core}} w(z) \cdot (d(p,z) + d(z,q)) \right) \cdot \epsilon^2 \beta^2 n^4 \cdot w(p) \cdot w(q)}{64 \cdot \omega_p \cdot \omega_q} \\ &= \frac{\epsilon^2 \beta^2 n^3}{64} \cdot \left( \frac{\omega_p}{w(p)} + \frac{\omega_q}{w(q)} \right) \cdot \frac{w(p) \cdot w(q)}{\omega_p \cdot \omega_q} \\ &\leq \frac{\epsilon^2 \beta^2 n^3}{64} \cdot \frac{2}{\min\{\frac{\omega_p}{w(p)}, \frac{\omega_q}{w(q)}\}} \leq \frac{\epsilon^2 \beta^2 n^3}{64} \cdot \frac{4}{n} = \frac{\epsilon^2 \beta^2 n^2}{16} \end{aligned}$$

We conclude

$$\max_{p',q' \in X}(e_{p'q'}) \leq \frac{1}{(1-\epsilon)^2} \cdot 4 \cdot \text{avg}_{p',q' \in X}(e_{p'q'}) \leq 16 \cdot \text{avg}_{p',q' \in X}(e_{p'q'}) \ .$$

□

Since $W$ is a dense graph we can apply the algorithm of [51] to find a $(1 \pm \epsilon)$-approximate MaxCut $(L', R')$ for $W$ in $O\left(|X|^2 \cdot 2^{((1/\epsilon)^{O(1)})}\right) = O\left(\log^2 n \cdot 2^{((1/\epsilon)^{O(1)})}\right)$ time. We extrapolate this cut to a $(1 \pm \epsilon)^3 = (1 \pm O(\epsilon))$-approximate Maxcut $(L, R)$ on $P_{core}$ and get the following result:

**Theorem 16** *A $(1+\epsilon)$-approximate MaxCut $(L, R)$ on $P_{core}$ can be found in time*

$$O\left(\log^2 n \cdot 2^{((1/\epsilon)^{O(1)})}\right) \ .$$

**Remark 5.5.12** *We can even compute in the same time an implicit MaxCut $(A, B)$ of the input point set $P$ from the coreset:*

*After computing a good cut $(L, R)$ of $P_{core}$ we can provide a partition of the whole space $[0,1]^d$ into two parts $\mathcal{L}$ and $\mathcal{R}$. During our coreset construction we store the information about the mappings of points to coreset points. When the points of a cell $C$ are mapped to a coreset point $p \in L$ during the coreset construction, we assign the whole cell $C$ to $\mathcal{L}$. When the points of a cell $C$ are mapped to a coreset point $p \in R$ during the coreset construction, we assign the whole cell $C$ to $\mathcal{R}$. After the construction of the partition $(\mathcal{L}, \mathcal{R})$ of the plane we know that the partition $(A, B)$ of $P$ with $A := P \cap \mathcal{L}$ and $B := P \cap \mathcal{R}$ is a $(1+\epsilon)$-approximate MaxCut of $P$.*

*The computation of $(\mathcal{L}, \mathcal{R})$ from $(L, R)$ can be done in poly$(\log n, 1/\epsilon)$ time and space.*

Together with the results of Corollary 5.4.8 we obtain the fastest method published so far (in terms of $n$) to find a $(1 \pm \epsilon)$-approximation of the Euclidean MaxCut of a point set. Previous methods had runtime $O(n \cdot \log n \cdot (2^{(1/\epsilon)^{O(1)}} + \log n))$ [72] and $O(n^2 \cdot 2^{O(1/\epsilon^2)})$ [37].

**Theorem 17** *Given a set $P$ of $n$ points in $\mathbb{R}^d$, a $(1 \pm \epsilon)$-approximate solution for the Euclidean MaxCut of the points can be found in time $O(n \cdot \log(n/\epsilon) + \log^2 n \cdot \log\log n \cdot 2^{((1/\epsilon)^{O(1)})})$.*

### 5.5.4 MaxMatching

We are not aware of a method to compute an approximate MaxMatching solution on the weighted coreset points directly without expanding the coreset.

One could replace each point $p \in P_{core}$ having weight $w(p)$ by $w(p)$ unweighted points and run the algortihm of Gabow[45], which finds an exact best solution in $O(n^3)$ time.

### 5.5.5 MaxTSP

We are again not aware of a method to find a $(1 \pm \epsilon)$-approximation of the MaxTSP tour even on unweighted points. We could obtain a solution on the coreset $P_{core}$ by expanding the coreset to a point set of size $n$ and running an exhaustive search. The running time is $O(n!)$.

### 5.5.6 AverageDistance

The weighted average distance $\frac{1}{n^2}\sum_{p,q\in P_{core}} w(p)\cdot w(q)\cdot d(p,q)$ on the coreset $P_{core}$ can be easily computed in time $O(|P_{core}|^2) = O(\log^2 n\cdot \epsilon^{-d-2})$.

## 5.6 Coresets via Sampling

In the last sections we introduced a coreset construction technique, which is suitable to reduce the complexity of a huge point set. We showed that the huge point set can be replaced by a weighted point set of logarithmic size, which still holds all information needed to compute approximate solutions for various clustering problems. However, to construct the coreset we need access to the whole point set, which is not always given in real world applications dealing with huge point sets.

In this chapter we will alter the construction such that it depends only on point samples. We will show that the information about all point samples can itself be stored in polylogarithmic memory. This will help us to develop data streaming algorithms in Chapter 6. The point sample technique will also help us to maintain MaxCut clusterings of points when points are moving along linear trajectories. See Chapter 7 for details.

We still assume that all points lie in $[0,1]^d$ and that $Opt \geq 1/\widetilde{\Delta}$.

### 5.6.1 k-Median

We again consider $Z$ grids $\mathcal{G}_0,\ldots,\mathcal{G}_{Z-1}$ for $Z = \left\lceil \log\left(\frac{4\cdot k\cdot 10^d\cdot n(1+\log n)\cdot d^{(d+1)/2}\cdot\widetilde{\Delta}}{\epsilon^{d+1}}\right)\right\rceil + 1$, grid $\mathcal{G}_i$ having cell side length $\frac{1}{2^i}$. In each grid $\mathcal{G}_i$ we pick a random sample $S_i$ of points. To select our random sample we take every point with probability

$$p_i := \min\left\{\frac{\alpha}{\delta\cdot 2^i}, 1\right\}$$

into our sample $S_i$, where

$$\alpha = 6\cdot\epsilon^{-2}\ln(2\cdot Z\cdot 2^{Z\cdot d}/\rho)$$

and $\rho$ is the desired error probability of our algorithm. The sampling is done at least $\alpha$-wise independently, which means that for each set $A \subset P$ of at most $\alpha$ points and each partition $\{B,C\}$ of $A$:

$$\mathbf{Pr}[B\subset S_i \wedge C\cap S_i = \emptyset] = (p_i)^{|B|}\cdot(1-p_i)^{|C|}\ .$$

Essentially this means that for each subset $A\subset P$ of size $\alpha$ the sampling is done independently.

We will show that it follows from a variant of Chernoff bounds [113] that we can approximate the number of points in every heavy cell up to a multiplicative error of $(1\pm\epsilon)$ just using our point samples. The approximations will furthermore be good enough to detect the heavy cells and to construct an $\epsilon$-coreset in the same way as described before.

**Definition 5.6.1 (Considered as Heavy)** *For each cell $\mathcal{C}$ in grid $\mathcal{G}_i$ we define $n_{\mathcal{C}}$ as the number of points in $\mathcal{C}$. We define our estimation on the number of points as*

$$\widetilde{n_{\mathcal{C}}} := |S_i \cap \mathcal{C}| \cdot \frac{1}{p_i} \ .$$

*A cell $\mathcal{C}$ in grid $\mathcal{G}_i$ is* considered as heavy, *if*

$$\widetilde{n_{\mathcal{C}}} \geq (1-\epsilon) \cdot \delta \cdot 2^i \ .$$

**Lemma 5.6.2** *The following events hold with probability at least $1 - \rho/2$ for all grids $\mathcal{G}_i$ and each grid cell in $\mathcal{G}_i$:*

- *If $i \leq \log(\frac{2}{\epsilon\delta})$, then $\widetilde{n_{\mathcal{C}}} = n_{\mathcal{C}}$.*
- *If $\mathcal{C}$ contains at least $\delta \cdot 2^{i-1}$ points, then $(1-\epsilon) \cdot n_{\mathcal{C}} \leq \widetilde{n_{\mathcal{C}}} \leq (1+\epsilon) \cdot n_{\mathcal{C}}$.*
- *If $\mathcal{C}$ contains less than $\delta \cdot 2^{i-1}$ points, then $\widetilde{n_{\mathcal{C}}} < (1-\epsilon) \cdot \delta \cdot 2^i$ (and the cell $\mathcal{C}$ is not considered as heavy).*

**Proof :** If $i \leq \log(\frac{2}{\epsilon\delta})$ then $p_i = 1$, the sample set equals the point set, and $\widetilde{n_{\mathcal{C}}} = n_{\mathcal{C}}$.

Let $\mathcal{C}$ be an arbitrary grid cell in $\mathcal{G}_i$. To prove the last two statements for the single cell $\mathcal{C}$ we use Theorem 1 of Section 2.4.

For each point $p \in P$ let $X_p$ denote the indicator random variable for the event that $p \in S_i$. We want to show that $\sum_{p\in\mathcal{C}} X_p$ does not deviate much from its expectation. If a cell contains at least $\delta \cdot 2^{i-1}$ points then $\mathbf{E}[\sum_{p\in\mathcal{C}} X_p] \geq \alpha/2$. From Theorem 1 it follows:

$$\mathbf{Pr}\Big[\Big|\sum_{p\in\mathcal{C}} X_p - \mathbf{E}[\sum_{p\in\mathcal{C}} X_p]\Big| \geq \epsilon \cdot \mathbf{E}[\sum_{p\in\mathcal{C}} X_p]\Big] \leq e^{-\min\{\lfloor\alpha/2\rfloor, \lfloor\epsilon^2\alpha/6\rfloor\}}$$

Plugging in $\sum_{p\in\mathcal{C}} X_p = \widetilde{n_{\mathcal{C}}} \cdot p_i$ and $\mathbf{E}[\sum_{p\in\mathcal{C}} X_p] = n_{\mathcal{C}} \cdot p_i$ we obtain

$$\mathbf{Pr}\Big[\big|\widetilde{n_{\mathcal{C}}} - n_{\mathcal{C}}\big| \geq \epsilon \cdot n_{\mathcal{C}}\Big] \leq \frac{\rho}{2 \cdot Z \cdot 2^{Z\cdot d}} \ ,$$

and the second statement follows with probability $1 - \frac{\rho}{2\cdot Z\cdot 2^{Z\cdot d}}$.

Assume that $\mathcal{C}$ contains at most $\delta \cdot 2^{i-1}$ points. If $\mathcal{C}$ contains exactly $\delta \cdot 2^{i-1}$ points, we can conclude from the formula above that

$$\widetilde{n_{\mathcal{C}}} \leq (1+\epsilon) \cdot n_{\mathcal{C}} < (1+1/3) \cdot \delta \cdot 2^{i-1} = (1-1/3) \cdot \delta \cdot 2^i < (1-\epsilon) \cdot \delta \cdot 2^i$$

holds with probability $1 - \rho/(2 \cdot Z \cdot 2^{Zd})$. We observe that the distribution of $\widetilde{n_{\mathcal{C}}}$ displaces towards lower values when the number of points in the cell decreases, which means that $\mathbf{Pr}\big[\widetilde{n_{\mathcal{C}}} < (1-\epsilon)\delta \cdot 2^i\big] \geq 1 - \rho/(2 \cdot Z \cdot 2^{Zd})$ also holds for smaller numbers of points in $\mathcal{C}$.

We conclude that the two statements are valid for one fixed single cell $\mathcal{C}$ with probability $1-\rho/(2\cdot Z\cdot 2^{Zd})$. Since we have at most $Z$ grids, each grid having at most $2^{Z\cdot d}$ cells, the two statements are valid with probability $1-\rho/2$ for all cells in all grids by the union bound.

□

If $\widetilde{n_{\mathcal{C}}} \geq (1-\epsilon)\cdot\delta\cdot 2^i$, a cell $\mathcal{C}\in\mathcal{G}_i$ is considered as heavy. This way, we detect every heavy cell but we also consider some light cells as heavy.

We then compute a coreset by introducing a coreset point in each cell considered as heavy (as described in Section 5.2.1). This will increase the size of our coreset. The following corollaries show that the size of the coreset is still logarithmic in $n$.

**Corollary 5.6.3** *Assume that the statements of Lemma 5.6.2 hold for all cells in all grids (which happens with probability $1-\rho/2$).*

*If $\delta \geq \frac{\epsilon^{d+1}\cdot Opt}{8\cdot k\cdot 10^d\cdot(1+\log n)\cdot d^{(d+1)/2}}$, the size of the computed coreset is at most $\frac{33\cdot k\cdot 10^d\cdot(2+\log n)\cdot d^{d/2}}{\epsilon^{d+1}} = O(k\cdot\log n/\epsilon^{d+1})$.*

**Proof :** We can easily modify the proof of Lemma 5.2.8 by plugging in $\delta/2$ for the old value of $\delta$. The proof stays exactly the same and we can conclude that the size of the coreset is at most $\frac{4\cdot Opt}{\delta\cdot\sqrt{d}} + (\log n+2)\cdot k\cdot 3^d\cdot d^{d/2}$, which is smaller than the stated coreset size for $\delta \geq \frac{\epsilon^{d+1}\cdot Opt}{8\cdot k\cdot 10^d\cdot(1+\log n)\cdot d^{(d+1)/2}}$.

□

An important property of our sample technique is that although the sample can be large, it just occupies a small number of cells (and can, as we show later, be stored efficiently).

**Lemma 5.6.4** *Let $\delta \geq \frac{\epsilon^{d+1}\cdot Opt}{8\cdot k\cdot 10^d\cdot(1+\log n)\cdot d^{(d+1)/2}}$. Then we have points from at most*

$$\frac{193\cdot Z\cdot k\cdot 10^d\cdot(1+\log n)\cdot d^{d/2}\cdot\ln(2\cdot Z\cdot 2^{Zd}/\rho)}{\epsilon^{d+3}} = \widetilde{O}(k\cdot\log n\cdot\log^2(\widetilde{\Delta})\cdot\log(\rho^{-1})/\epsilon^{d+3})$$

*cells in the union of our sample sets with probability at least $1-\rho/2$.*

**Proof :** Let $\mathcal{G}_i$ be a fixed grid. We determine an upper bound on the number of points in non-center grid cells. Let us recall from the proof of Lemma 5.2.8 that every point except for those contained in the $k\cdot 3^d\cdot d^{d/2}$ center cells has a distance of at least $\frac{\sqrt{d}}{2^{i+1}}$ to the nearest center in an optimal solution. Thus the overall number of points in non-center cells is at most $\frac{Opt\cdot 2^{i+1}}{\sqrt{d}}$. Let $X_p$ denote the indicator random variable for the event that $p\in S_i$. Let $\mathcal{D}$ denote the set of non-center grid cells. We have $\mathbf{E}[\sum_{p\in\mathcal{D}} X_p] \leq p_i\cdot\frac{Opt\cdot 2^{i+1}}{\sqrt{d}} \leq \frac{2\cdot\alpha\cdot Opt}{\delta\cdot\sqrt{d}}$. We will assume $\mathbf{E}[\sum_{p\in\mathcal{D}} X_p] = \frac{2\cdot\alpha\cdot Opt}{\delta\cdot\sqrt{d}}$ as the distribution of $\sum_{p\in\mathcal{D}} X_p$ displaces towards lower values when $\mathbf{E}[\sum_{p\in\mathcal{D}} X_p] < \frac{2\cdot\alpha\cdot Opt}{\delta\cdot\sqrt{d}}$.

Applying Theorem 1 from Section 2.4 we get

$$\mathbf{Pr}\Big[\sum_{p\in\mathcal{D}} X_p \geq \frac{4\cdot\alpha\cdot Opt}{\delta\cdot\sqrt{d}}\Big] \leq \mathbf{Pr}\Big[\Big|\sum_{p\in\mathcal{D}} X_p - \mathbf{E}[\sum_{p\in\mathcal{D}} X_p]\Big| \geq \mathbf{E}[\sum_{p\in\mathcal{D}} X_p]\Big]$$
$$\leq e^{-\min\{\lfloor\alpha/2\rfloor,\lfloor\epsilon^2\frac{\alpha}{\delta}/3\rfloor\}} \leq \frac{\rho}{2Z} \quad .$$

Therefore, with probability at least $1-\rho/(2Z)$ we have at most $\frac{4\cdot\alpha\cdot Opt}{\delta\cdot\sqrt{d}}$ points from non-center cells in our sample. If no two of these points are contained in the same grid cell we get an upper bound of $\frac{4\cdot\alpha\cdot Opt}{\delta\cdot\sqrt{d}}$ on the number of non-center cells that contain a sample point. Since there are at most $k\cdot 3^d\cdot d^{d/2}$ center cells the number of cells occupied by sample points is at most $\frac{4\cdot\alpha\cdot Opt}{\delta\cdot\sqrt{d}}+k\cdot 3^d\cdot d^{d/2}\leq\frac{193\cdot k\cdot 10^d\cdot(1+\log n)\cdot d^{d/2}\cdot\ln(2\cdot Z\cdot 2^{Zd}/\rho)}{\epsilon^{d+3}}$.

Since these arguments hold for each of the $Z$ grids with probability $1-\rho/(2Z)$, the Lemma follows from the union bound. □

To obtain a coreset we use the estimations $\widetilde{n_{\mathcal{C}}}$ of the number of points in cells to identify the cells we consider as heavy (all cells having $\widetilde{n_{\mathcal{C}}}\geq(1-\epsilon)\delta\,2^i$). Since all heavy cells are considered as heavy we obtain a finer coreset than before. We will now show how to find a good assignment of weights to the computed coreset points, such that the computed coreset is an $\epsilon$-coreset for $P$.

Since the weight of a coreset point will also depend on the number of points in some light cells, we have to estimate the number of points in these cells. To get an estimate for all required cells we use the following procedure. We require that the estimate $\widetilde{n_{\mathcal{C}}}$ for the number of points in a cell considered as heavy is a $(1\pm\epsilon)$-approximation and that in every cell $\mathcal{C}\in\mathcal{G}_i$ considered as light there are not more than $\delta\cdot 2^i$ points (our coreset construction uses only these assumptions, and they hold according to Lemma 5.6.2 with probability $1-\rho/2$).

We call a cell *useful*, if it is either considered as heavy or a direct subcell of a cell considered as heavy. We have to deal with the fact that the sum of the total estimated number of points $\sum_{\mathcal{C}_i\text{ subcell of }\mathcal{C}}\widetilde{n_{\mathcal{C}_i}}$ in the subcells of $\mathcal{C}$ can exceed the estimated number of points $\widetilde{n_{\mathcal{C}}}$ in $\mathcal{C}$. To avoid this we have to compute new integral estimates $E_{\mathcal{C}}$ for the number of points in each *useful* cell $\mathcal{C}$, which still have the guarantee to be near the real value $n_{\mathcal{C}}$ and which are consistent with the values $E_{\mathcal{C}_i}$ of the subcells of $E_{\mathcal{C}}$. We do this by first computing upper and lower bounds $U_{\mathcal{C}}$ resp. $L_{\mathcal{C}}$ on $n_{\mathcal{C}}$ for all useful cells. We will then adjust these bounds to be consistent with the bounds for the subcells. Finally we will use the bounds to compute new estimates $E_{\mathcal{C}}$.

For $i>\log(\frac{2}{\epsilon\delta})$ and every cell $C\in\mathcal{G}_i$ considered as heavy we define $L_{\mathcal{C}}=\lceil\widetilde{n_{\mathcal{C}}}/(1+\epsilon)\rceil$ and $U_{\mathcal{C}}=\lfloor\widetilde{n_{\mathcal{C}}}/(1-\epsilon)\rfloor$. For $i>\log(\frac{2}{\epsilon\delta})$ and every cell $\mathcal{C}\in\mathcal{G}_i$ considered as light we define $L_{\mathcal{C}}=0$ and $U_{\mathcal{C}}=\lfloor\delta 2^i\rfloor$. For $i\leq\log(\frac{2}{\epsilon\delta})$ and every cell $C\in\mathcal{G}_i$ we define $L_{\mathcal{C}}=\widetilde{n_{\mathcal{C}}}$ and $U_{\mathcal{C}}=\widetilde{n_{\mathcal{C}}}$ (since we know the number of points in $\mathcal{C}$ exactly). Using these definitions we know for every cell that $L_{\mathcal{C}}\leq n_{\mathcal{C}}\leq U_{\mathcal{C}}$).

The estimates $E_{\mathcal{C}}$ can be computed bottom-up by adjusting the bounds $L_{\mathcal{C}}$ and $U_{\mathcal{C}}$ in cases of conflicts:

We first compute new lower and upper bounds $L_{\mathcal{C}}$ and $U_{\mathcal{C}}$ for all useful cells bottom-up. We look at the smallest cell $\mathcal{C}$ considered as heavy. Let $\mathcal{C}_i, i\in\{1,...,2^d\}$ be its subcells. If $\sum_{i=1}^{2^d}L_{\mathcal{C}_i}>L_{\mathcal{C}}$, we set $L_{\mathcal{C}}:=\sum_{i=1}^{2^d}L_{\mathcal{C}_i}$. If $\sum_{i=1}^{2^d}U_{\mathcal{C}_i}<U_{\mathcal{C}}$, we set $U_{\mathcal{C}}:=\sum_{i=1}^{2^d}U_{\mathcal{C}_i}$. After the assignment $L_{\mathcal{C}}\leq n_{\mathcal{C}}\leq U_{\mathcal{C}}$ still holds. We use this technique for all cells considered as heavy (in the order of increasing size), getting better bounds $L_{\mathcal{C}}$ and $U_{\mathcal{C}}$. From these bounds we then compute the values $E_{\mathcal{C}}$ top-down. Since the bounds $L_{\mathcal{C}}$ and $U_{\mathcal{C}}$ are always at least as strong as the bounds of the subcells, we can always easily find integral values $E_{\mathcal{C}}$ satisfying $L_{\mathcal{C}}\leq E_{\mathcal{C}}\leq U_{\mathcal{C}}$ and $\sum_{i=1}^{2^d}E_{\mathcal{C}_i}=E_{\mathcal{C}}$.

**Corollary 5.6.5** *Assume that the statements of Lemma* 5.6.2 *are true for all grids and all cells.*

*Then for each cell $\mathcal{C}$ identified as heavy we have* $(1-4\epsilon)n_{\mathcal{C}} \leq E_{\mathcal{C}} \leq (1+4\epsilon)n_{\mathcal{C}}$.
*For each cell* $\mathcal{C} \in \mathcal{G}_i$ *with* $i \leq \log(\frac{2}{\epsilon\delta})$ *we have* $E_{\mathcal{C}} = n_{\mathcal{C}}$.
*All estimates* $E_{\mathcal{C}}$ *are integral and consistent with the estimates* $E_{\mathcal{C}_i}$ *for the subcells* $\mathcal{C}_i$ *of* $\mathcal{C}$.

**Proof :** The claim follows directly from the following two sequences of inequalities.

$$E_{\mathcal{C}} \;\geq\; L_{\mathcal{C}} \;\geq\; \widetilde{n_{\mathcal{C}}}/(1+\epsilon) \;\geq\; \frac{1-\epsilon}{1+\epsilon} n_{\mathcal{C}} \;\geq\; (1-2\epsilon)n_{\mathcal{C}}$$

and

$$E_{\mathcal{C}} \;\leq\; U_{\mathcal{C}} \;\leq\; \widetilde{n_{\mathcal{C}}}/(1-\epsilon) \;\leq\; \frac{1+\epsilon}{1-\epsilon} n_{\mathcal{C}} \;\leq\; (1+4\epsilon)n_{\mathcal{C}} \;.$$

□

We now apply the algorithm described in Section 5.2 to our estimations $E_{\mathcal{C}}$ and compute a coreset.

**Lemma 5.6.6** *If* $\delta \leq \frac{\epsilon^{d+1} \cdot Opt}{4 \cdot k \cdot 10^d \cdot (1+\log n) \cdot d^{(d+1)/2}}$ *and* $\epsilon < 1/15$*, the coreset computed with respect to the values* $E_{\mathcal{C}}$ *is a* $11\epsilon$*-coreset of* $P$ *with probability* $1-\rho$.

**Proof :** Let $P'$ be a point set that is distributed according to our estimations $E_{\mathcal{C}}$ (so for every useful cell C we have $|P' \cap \mathcal{C}| = E_{\mathcal{C}}$). The proof of Lemma 5.2.5 shows that the coreset computed by our algorithm is an $\epsilon$-coreset for $P'$. Let $Q = \{q_1, \ldots, q_m\}$ be the computed coreset points. We will show that if we would know the point sets $P$ and $P'$, we could (using the coreset method) compute mappings $\gamma : P \to Q$ and $\gamma' : P' \to Q$ and corresponding weight functions $w : Q \to \mathbb{N}$ and $w' : Q \to \mathbb{N}$, such that $(Q, w)$ is an $\epsilon$-coreset for $P$ and $(Q, w')$ is an $\epsilon$-coreset for $P'$ and for all $q_i \in Q$ we have:

$$(1-4\epsilon)(w(q_i)-1) \leq w'(q_i) \leq (1+4\epsilon)w(q_i)+1 \tag{5.2}$$

and

$$w(q_i) \leq \frac{1}{\epsilon} \;\Longrightarrow\; w(q_i) = w'(q_i) \;. \tag{5.3}$$

From that we will conclude that each solution on the point set $P'$ differs by at most a factor of $(1+O(\epsilon))$ from the solution on the point set $P$. Since the computed coreset is an $\epsilon$-coreset for $P'$ it follows that it is a $O(\epsilon)$-coreset for $P$.

Let us construct the mappings $\gamma$ and $\gamma'$. Lemma 5.2.5 shows that we construct a $(1+\epsilon)$-coreset when we map each point $p$ to a coreset point in the smallest heavy cell it is contained in. We start the assignment of points to coreset points within the smallest useful cells. Since the smallest useful cells are not heavy we do not assign any points to them. We proceed to assign points in the useful cells at the next higher level. Going through the levels bottom-up we will assign all points in useful cells and maintain the invariants (5.2) and (5.3).

Let $\mathcal{C}$ be a cell considered as heavy. If there is no subcell considered as heavy, the algorithm introduces a new coreset point $q$. We map all $E_{\mathcal{C}}$ points from $P'$ to $q$ and all $n_{\mathcal{C}}$ points from $P$ to $q$. Then $w(q) = E_{\mathcal{C}}$ and $w'(q) = n_{\mathcal{C}}$. Notice that $w(q) < \frac{1}{\epsilon}$ can only happen for a coreset point $q$ when a cell in grid $\mathcal{G}_i$ or a subcell is considered as heavy. Then $\delta \cdot 2^{i-1} < \frac{1}{\epsilon}$ according to Lemma 5.6.2. This only happens in grids $\mathcal{G}_i$ with $i < \log(\frac{2}{\epsilon\delta})$ where $E_{\mathcal{C}} = n_{\mathcal{C}}$ (Corollary 5.6.5) and therefore $w(q) = w'(q)$ after the assignment. This shows that invariant (5.3) holds. Invariant (5.2) follows directly from Corollary 5.6.5.

Let us now consider the case that $\mathcal{C} \in \mathcal{G}_i$ has already $c$ coreset points $q_1, \ldots, q_c \in Q$ with weights $w(q_i)$ and $w'(q_i)$, respectively and let us assume that the invariants (5.2) and (5.3) hold for all these coreset points. Let $l := n_{\mathcal{C}} - \sum_{i=1}^{c} w(q_i)$ resp. $l' := E_{\mathcal{C}} - \sum_{i=1}^{c} w'(q_i)$ be the number of points which have to be assigned to the coreset points $q_i$ by $\gamma$ resp. $\gamma'$.

We consider six cases:

- $l = 0$ *and* $l' = 0$: In this case nothing has to be assigned and the invariant holds by the assumption of the induction step.

- $l > 0$ *and* $l' = 0$ *and* $i \geq \log(\frac{2}{\epsilon\delta})$: Each cell considered as heavy has at least $\delta \cdot 2^{i-1}$ points accordings to Lemma 5.6.2 (The threshold can only have been higher during the coreset constructions so far). Therefore each coreset point must have a weight of at least $\delta \cdot 2^{i-1} \geq \frac{1}{\epsilon}$. It remains to show invariant (5.2). We have

$$(1-4\epsilon)\sum_{i=1}^{c} w(q_i) \;=\; (1-4\epsilon)(n_{\mathcal{C}} - l) \;<\; (1-4\epsilon)n_{\mathcal{C}} \;\leq\; E_{\mathcal{C}} = \sum_{i=1}^{c} w'(q_i)\ .$$

  Therefore for one $q_i$ we have $(1-4\epsilon)w(q_i) < w'(q_i)$ and we can assign at least one point from $P$ to $q_i$ by $\gamma$ without violating invariant (5.2). After that assignment either $l = 0$ or we find again a $q_i$ we can assign points to. We go on with this assignment until $l = 0$.

- $l = 0$ *and* $l' > 0$ *and* $i \geq \log(\frac{2}{\epsilon\delta})$: Again we only have to show invariant (5.2). We have

$$\begin{aligned}\sum_{i=1}^{c} w'(q_i) \;&=\; E_{\mathcal{C}} - l' \;<\; E_{\mathcal{C}} \\ &\leq\; (1+4\epsilon)n_{\mathcal{C}} \;=\; (1+4\epsilon)(n_{\mathcal{C}} - l) = (1+4\epsilon)\sum_{i=1}^{c} w(q_i)\ .\end{aligned}$$

  Therefore for one $q_i$ we have $w'(q_i) < (1+4\epsilon)w(q_i)$ and we can assign at least one point from $P'$ to $q_i$ by $\gamma'$ without violating the invariant. After that assignment either $l' = 0$ or we again find a $q_i$ we can assign points to. We go on with this assignment until $l' = 0$.

- $l > 0$ *and* $l' = 0$ *and* $i < \log(\frac{2}{\epsilon\delta})$: In this case we have $E_{\mathcal{C}} = n_{\mathcal{C}}$ and

$$\sum_{i=1}^{c} w(q_i) \;=\; n_{\mathcal{C}} - l \;<\; n_{\mathcal{C}} \;=\; E_{\mathcal{C}} = \sum_{i=1}^{c} w'(q_i)\ .$$

Since all $w(q_i)$ and $w'(q_i)$ are integral, we have $w(q_i) \leq w'(q_i) - 1$ for one $q_i$ and we can assign at least one point from $P$ to $q_i$ by $\gamma$ without violating the invariants. After that assignment either $l = 0$ or we find again a $q_i$ we can assign points to. We go on with this assignment until $l = 0$.

- $l = 0$ *and* $l' > 0$ *and* $i < \log(\frac{2}{\epsilon\delta})$: Again we have $E_{\mathcal{C}} = n_{\mathcal{C}}$ and

$$\sum_{i=1}^{c} w'(q_i) \;=\; E_{\mathcal{C}} - l' \;<\; E_{\mathcal{C}} \;=\; n_{\mathcal{C}} \;=\; (n_{\mathcal{C}} - l) = \sum_{i=1}^{c} w(q_i) \;\; .$$

Therefore for one $q_i$ we have $w'(q_i) \leq w(q_i) - 1$ and can assign at least one point from $P'$ to $q_i$ by $\gamma'$ without violating the invariant. After that assignment either $l' = 0$ or we again find a $q_i$ we can assign points to. We go on with this assignment until $l' = 0$.

- $l > 0$ *and* $l' > 0$: We assign $\min\{l, l'\}$ points from $P$ to $q_1$ by $\gamma$ and $\min\{l, l'\}$ points from $P'$ to $q_1$ by $\gamma'$. This does not violate the invariant. After the assignment we are in one of the other cases.

After the inductive assignment we have constructed mappings $\gamma$ and $\gamma'$ and corresponding weight functions $w, w'$, such that invariants (5.2) and (5.3) hold. Using the invariants it is easy to show that for all coreset points $q$:

$$(1 - 5\epsilon)w(q) \leq w'(q) \leq (1 + 5\epsilon)w(q) \;\; . \tag{5.4}$$

If $w(q) \leq \frac{1}{\epsilon}$ the inequality follows from invariant (5.3). If $w(q) > \frac{1}{\epsilon}$ we have:

$$w'(q) \geq (1-4\epsilon)(w(q)-1) \geq (1-4\epsilon)(w(q)-\epsilon\cdot w(q)) = (1-4\epsilon-\epsilon+4\epsilon^2)w(q) \geq (1-5\epsilon)w(q)$$

and

$$w'(q) \leq (1 + 4\epsilon)w(q) + 1 \leq (1 + 4\epsilon)w(q) + \epsilon \cdot w(q) = (1 + 5\epsilon)w(q)$$

and inequality (5.4) follows.

Let $A$ denote the coreset computed by our sample algorithm. Since $A$ is an $\epsilon$-coreset for $P'$ we know for each set of centers $C$:

$$\mathit{Median}(A, C) \in (1 \pm \epsilon) \cdot \mathit{Median}(P', C) \;\; .$$

From the arguments above we know that

$$\mathit{Median}(P', C) \in \frac{1}{1 \pm \epsilon} \cdot \mathit{Median}((Q, w'), C) \subset (1 \pm 2\epsilon) \cdot \mathit{Median}((Q, w'), C) \;\; .$$

Since the weights $w'(q)$ and $w(q)$ of each coreset-point in $q \in Q$ differ by at most $5\epsilon \cdot w(q)$, we can conclude:

$$\mathit{Median}((Q, w'), C) \in (1 \pm 5\epsilon) \cdot \mathit{Median}((Q, w), C) \;\; .$$

Since $(Q, \gamma)$ is an $\epsilon$-coreset for $P$, we obtain:

$$\mathit{Median}((Q, w), C) \in (1 \pm \epsilon) \cdot \mathit{Median}(P, C) \ .$$

Alltogether we get for $\epsilon < 1/15$ :

$$\mathit{Median}(A, C) \in (1 \pm \epsilon)^2 \cdot (1 \pm 2\epsilon) \cdot (1 \pm 5\epsilon) \cdot \mathit{Median}(P, C) \subset (1 \pm 11\epsilon) \cdot \mathit{Median}(P, C) \ .$$

□

### 5.6.2 k-Means

In this section we will adapt the sampling technique of the last section to the problem k-means. The proofs will be very similar to the proofs of the last section.

We again consider $Z$ nested grids $\mathcal{G}_0, \ldots, \mathcal{G}_{Z-1}$ for $Z = \left\lceil \log\left(\frac{8 \cdot k \cdot 33^{d+1} \cdot n(1+\log n) \cdot d^{(d+2)/2} \cdot \tilde{\Delta}}{\epsilon^{d+2}}\right)\right\rceil + 1$, grid $\mathcal{G}_i$ having cell side length $\frac{1}{2^i}$. In each grid $\mathcal{G}_i$ we pick a random sample $S_i$ of points. To select our random sample we take every point with probability

$$p_i = \min\left\{\frac{\alpha}{\delta \cdot 4^i}, 1\right\}$$

into our sample $S_i$, where

$$\alpha = 6 \cdot \epsilon^{-2} \ln(2 \cdot Z \cdot 2^{Z \cdot d}/\rho)$$

and $\rho$ is the desired error probability of our algorithm. The sampling is done at least $\alpha$-wise independently, which means that for each set $A \subset P$ of at most $\alpha$ points and each partition $\{B, C\}$ of $A$:

$$\mathbf{Pr}[B \subset S_i \wedge C \cap S_i = \emptyset] = (p_i)^{|B|} \cdot (1 - p_i)^{|C|} \ .$$

Essentially this means that for each subset $A \subset P$ of size $\alpha$ the sampling is done independently.

We will show that it follows from a variant of Chernoff bounds [113] that we can approximate the number of points in every heavy cell up to a multiplicative error of $(1 \pm \epsilon)$ just using our point samples. The approximations will furthermore be good enough to detect the heavy cells and to construct an $\epsilon$-coreset in the same way as described before.

**Definition 5.6.7 (Considered as Heavy)** *For each cell $\mathcal{C}$ in grid $\mathcal{G}_i$ we define $n_{\mathcal{C}}$ as the number of points in $\mathcal{C}$. We define our estimation on the number of points as*

$$\widetilde{n_{\mathcal{C}}} := |S_i \cap \mathcal{C}| \cdot \frac{1}{p_i} \ .$$

*A cell $\mathcal{C}$ in grid $\mathcal{G}_i$ is* considered as heavy*, if*

$$\widetilde{n_{\mathcal{C}}} \geq (1 - \epsilon) \cdot \delta \cdot 4^i \ .$$

**Lemma 5.6.8** *The following events hold with probability at least $1 - \rho/2$ for all grids $\mathcal{G}_i$ and each grid cell in $\mathcal{G}_i$:*

- *If* $i \leq \log_4(\frac{2}{\epsilon\delta})$, *then* $\widetilde{n_{\mathcal{C}}} = n_{\mathcal{C}}$.
- *If* $\mathcal{C}$ *contains at least* $\delta \cdot 4^i/2$ *points, then* $(1-\epsilon) \cdot n_{\mathcal{C}} \leq \widetilde{n_{\mathcal{C}}} \leq (1+\epsilon) \cdot n_{\mathcal{C}}$.
- *If* $\mathcal{C}$ *contains less than* $\delta \cdot 4^i/2$ *points, then* $\widetilde{n_{\mathcal{C}}} < (1-\epsilon) \cdot \delta \cdot 4^i$ *(and the cell* $\mathcal{C}$ *is not considered as heavy).*

**Proof :** If $i \leq \log_4(\frac{2}{\epsilon\delta})$ then $p_i = 1$, the sample set equals the point set, and $\widetilde{n_{\mathcal{C}}} = n_{\mathcal{C}}$.

Let $\mathcal{C}$ be an arbitrary grid cell in $\mathcal{G}_i$. To prove the last two statements for the single cell $\mathcal{C}$ we use Theorem 1 of Section 2.4.

For each point $p \in P$ let $X_p$ denote the indicator random variable for the event that $p \in S_i$. We want to show that $\sum_{p\in\mathcal{C}} X_p$ does not deviate much from its expectation. If a cell contains at least $\delta \cdot 4^i/2$ points then $\mathbf{E}[\sum_{p\in\mathcal{C}} X_p] \geq \alpha/2$. From Theorem 1 it follows:

$$\mathbf{Pr}\Big[\Big|\sum_{p\in\mathcal{C}} X_p - \mathbf{E}[\sum_{p\in\mathcal{C}} X_p]\Big| \geq \epsilon \cdot \mathbf{E}[\sum_{p\in\mathcal{C}} X_p]\Big] \leq e^{-\min\{\lfloor\alpha/2\rfloor, \lfloor\epsilon^2\alpha/6\rfloor\}}$$

Plugging in $\sum_{p\in\mathcal{C}} X_p = \widetilde{n_{\mathcal{C}}} \cdot p_i$ and $\mathbf{E}[\sum_{p\in\mathcal{C}} X_p] = n_{\mathcal{C}} \cdot p_i$ we obtain

$$\mathbf{Pr}\Big[\big|\widetilde{n_{\mathcal{C}}} - n_{\mathcal{C}}\big| \geq \epsilon \cdot n_{\mathcal{C}}\Big] \leq \frac{\rho}{2 \cdot Z \cdot 2^{Z\cdot d}} \ ,$$

and the second statement follows with probability $1 - \frac{\rho}{2\cdot Z\cdot 2^{Z\cdot d}}$.

Assume that $\mathcal{C}$ contains at most $\delta \cdot 4^i/2$ points. If $\mathcal{C}$ contains exactly $\delta \cdot 4^i/2$ points, we can conclude from the formula above that

$$\widetilde{n_{\mathcal{C}}} \leq (1+\epsilon) \cdot n_{\mathcal{C}} < (1+1/3) \cdot \delta \cdot 4^i/2 = (1-1/3) \cdot \delta \cdot 4^i < (1-\epsilon) \cdot \delta \cdot 4^i$$

holds with probability $1 - \rho/(2 \cdot Z \cdot 2^{Zd})$. We observe that the distribution of $\widetilde{n_{\mathcal{C}}}$ displaces towards lower values when the number of points in the cell decreases, which means that $\mathbf{Pr}\big[\widetilde{n_{\mathcal{C}}} < (1-\epsilon)\delta \cdot 4^i\big] \geq 1 - \rho/(2 \cdot Z \cdot 2^{Zd})$ also holds for smaller numbers of points in $\mathcal{C}$.

We conclude that the two statements are valid for one fixed single cell $\mathcal{C}$ with probability $1 - \rho/(2 \cdot Z \cdot 2^{Zd})$. Since we have at most $Z$ grids, each grid having at most $2^{Z\cdot d}$ cells, the two statements are valid with probability $1 - \rho/2$ for all cells in all grids by the union bound.

□

If $\widetilde{n_{\mathcal{C}}} \geq (1-\epsilon) \cdot \delta \cdot 4^i$, a cell $\mathcal{C} \in \mathcal{G}_i$ is considered as heavy. This way, we detect every heavy cell but we also consider some light cells as heavy.

We then compute a coreset by introducing a coreset point in each cell considered as heavy (as described in Section 5.3.1). This will increase the size of our coreset. The following corollaries show that the size of the coreset is still logarithmic in $n$.

**Corollary 5.6.9** *Assume that the statements of Lemma 5.6.8 hold for all cells in all grids.*
*If* $\delta \geq \frac{\epsilon^{d+2}\cdot Opt}{16\cdot k\cdot 33^{d+1}\cdot(1+\log n)\cdot d^{(d+2)/2}}$, *the size of the computed coreset is at most*

$$\frac{129 \cdot k \cdot 33^{d+1} \cdot (1+\log n) \cdot d^{d/2}}{\epsilon^{d+2}} = O(k \cdot \log n/\epsilon^{d+2}) \ .$$

**Proof :** We can easily modify the proof of Lemma 5.3.8 by plugging in $\delta/2$ for the old value of $\delta$. The proof stays exactly the same and we can conclude that the size of the coreset is at most $\frac{8 \cdot Opt}{\delta \cdot d} + (\log n + 1) \cdot k \cdot 3^d \cdot d^{d/2}$, which is smaller than the stated coreset size for $\delta \geq \frac{\epsilon^{d+2} \cdot Opt}{16 \cdot k \cdot 33^{d+1} \cdot (1+\log n) \cdot d^{(d+2)/2}}$.

□

An important property of our sample technique is that although the sample can be large, it just occupies a small number of cells (and can, as we show later, be stored efficiently).

**Lemma 5.6.10** *Let* $\delta \geq \frac{\epsilon^{d+2} \cdot Opt}{16 \cdot k \cdot 33^{d+1} \cdot (1+\log n) \cdot d^{(d+2)/2}}$. *Then we have points from at most*

$$\frac{769 \cdot Z \cdot k \cdot 33^{d+1} \cdot (1 + \log n) \cdot d^{d/2} \cdot \ln(2 \cdot Z \cdot 2^{Zd}/\rho)}{\epsilon^{d+4}} = \widetilde{O}(k \cdot \log n \cdot \log^2(\widetilde{\Delta}) \cdot \log(\rho^{-1})/\epsilon^{d+4})$$

*cells in the union of our sample sets with probability at least* $1 - \rho/2$.

**Proof :** Let $\mathcal{G}_i$ be a fixed grid. We determine an upper bound on the number of points in non-center grid cells. Let us recall from the proof of Lemma 5.3.8 that every point except for those contained in the $k \cdot 3^d \cdot d^{d/2}$ center cells has a distance of at least $\frac{\sqrt{d}}{2^{i+1}}$ to the nearest center in an optimal solution and therefore contributes with at least $\frac{d}{4^{i+1}}$ to the optimal solution. Thus the overall number of points in non-center cells is at most $\frac{Opt \cdot 4^{i+1}}{d}$. Let $X_p$ denote the indicator random variable for the event that $p \in S_i$. Let $\mathcal{D}$ denote the set of non-center grid cells. We have $\mathbf{E}[\sum_{p \in \mathcal{D}} X_p] \leq p_i \cdot \frac{Opt \cdot 4^{i+1}}{d} \leq \frac{4 \cdot \alpha \cdot Opt}{\delta \cdot d}$. We will assume $\mathbf{E}[\sum_{p \in \mathcal{D}} X_p] = \frac{4 \cdot \alpha \cdot Opt}{\delta \cdot d}$ as the distribution of $\sum_{p \in \mathcal{D}} X_p$ displaces towards lower values when $\mathbf{E}[\sum_{p \in \mathcal{D}} X_p] < \frac{4 \cdot \alpha \cdot Opt}{\delta \cdot d}$.

Applying Theorem 1 of Section 2.4 we get

$$\mathbf{Pr}\Big[\sum_{p \in \mathcal{D}} X_p \geq \frac{8 \cdot \alpha \cdot Opt}{\delta \cdot d}\Big] \leq \mathbf{Pr}\Big[\Big|\sum_{p \in \mathcal{D}} X_p - \mathbf{E}[\sum_{p \in \mathcal{D}} X_p]\Big| \geq \mathbf{E}[\sum_{p \in \mathcal{D}} X_p]\Big]$$
$$\leq e^{-\min\{\lfloor \alpha/2 \rfloor, \lfloor \epsilon^2 \frac{\alpha}{\delta}/3 \rfloor\}} \leq \frac{\rho}{2Z} \quad .$$

Therefore, with probability at least $1 - \rho/(2Z)$ we have at most $\frac{8 \cdot \alpha \cdot Opt}{\delta \cdot d}$ points from non-center cells in our sample. Since in the worst case no two of these points are contained in the same grid cell we get an upper bound of $\frac{8 \cdot \alpha \cdot Opt}{\delta \cdot d}$ on the number of non-center cells that contain a sample point. Since there are at most $k \cdot 3^d \cdot d^{d/2}$ center cells the number of cells occupied by sample points is at most $\frac{8 \cdot \alpha \cdot Opt}{\delta \cdot d} + k \cdot 3^d \cdot d^{d/2} \leq \frac{769 \cdot k \cdot 33^{d+1} \cdot (1+\log n) \cdot d^{d/2} \cdot \ln(2 \cdot Z \cdot 2^{Zd}/\rho)}{\epsilon^{d+4}}$.

Since these arguments hold for each of the $Z$ grids with probability $1 - \rho/(2Z)$, the Lemma follows from the union bound. □

To obtain a coreset we use the estimations $\widetilde{n_c}$ of the number of points in heavy cells to identify the cells we consider as heavy (all cells having $\widetilde{n_c} \geq (1 - \epsilon)\delta 4^i$). Since all heavy cells are considered as heavy we obtain a finer coreset than before. We will now show how to find a good assignment of weights to the computed coreset points, such that the computed coreset is an $\epsilon$-coreset for $P$.

Since the weight of a coreset point will also depend on the number of points in some light cells, we have to estimate the number of points in these cells. To get an estimate for all required cells we use the following procedure. We require that the estimate $\widetilde{n_{\mathcal{C}}}$ for the number of points in a heavy cell is a $(1 \pm \epsilon)$-approximation and that in every cell $\mathcal{C} \in \mathcal{G}_i$ considered as light there are not more than $\delta \cdot 4^i$ points (our coreset construction uses only these assumptions, and they hold according to Lemma 5.6.8 with probability $1 - \rho/2$).

We call a cell *useful*, if it is either considered as heavy or a direct subcell of a cell considered as heavy. We have to deal with the fact that the sum of the total estimated number of points $\sum_{\mathcal{C}_i \text{ subcell of } \mathcal{C}} \widetilde{n_{\mathcal{C}_i}}$ in the subcells of $\mathcal{C}$ can exceed the estimated number of points $\widetilde{n_{\mathcal{C}}}$ in $\mathcal{C}$. To avoid this we have to compute new integral estimates $E_{\mathcal{C}}$ for the number of points in each *useful* cell $\mathcal{C}$, which still have the guarantee to be near the real value $n_{\mathcal{C}}$ and which are consistent with the values $E_{\mathcal{C}_i}$ of the subcells of $E_{\mathcal{C}}$. We do this by first computing upper and lower bounds $U_{\mathcal{C}}$ resp. $L_{\mathcal{C}}$ on $n_{\mathcal{C}}$ for all useful cells. We will then adjust these bounds to be consistent with the bounds for the subcells. Finally we will use the bounds to compute new estimates $E_{\mathcal{C}}$.

For $i > \log_4(\frac{2}{\epsilon\delta})$ and every cell $C \in \mathcal{G}_i$ considered as heavy we define $L_{\mathcal{C}} = \lceil \widetilde{n_{\mathcal{C}}}/(1+\epsilon) \rceil$ and $U_{\mathcal{C}} = \lfloor \widetilde{n_{\mathcal{C}}}/(1-\epsilon) \rfloor$. For $i > \log_4(\frac{2}{\epsilon\delta})$ and every cell $\mathcal{C} \in \mathcal{G}_i$ considered as light we define $L_{\mathcal{C}} = 0$ and $U_{\mathcal{C}} = \lfloor \delta 4^i \rfloor$. For $i \leq \log_4(\frac{2}{\epsilon\delta})$ and every cell $C \in \mathcal{G}_i$ we define $L_{\mathcal{C}} = \widetilde{n_{\mathcal{C}}}$ and $U_{\mathcal{C}} = \widetilde{n_{\mathcal{C}}}$ (since we know the number of points in $\mathcal{C}$ exactly). Using these definitions we know for every cell that $L_{\mathcal{C}} \leq n_{\mathcal{C}} \leq U_{\mathcal{C}}$).

The estimates $E_{\mathcal{C}}$ can be computed bottom-up by adjusting the bounds $L_{\mathcal{C}}$ and $U_{\mathcal{C}}$ in cases of conflicts:

We first compute new lower and upper bounds $L_{\mathcal{C}}$ and $U_{\mathcal{C}}$ for all useful cells bottom-up. We look at the smallest cell $\mathcal{C}$ considered as heavy. Let $\mathcal{C}_i, i \in \{1, ..., 2^d\}$ be its subcells. If $\sum_{i=1}^{2^d} L_{\mathcal{C}_i} > L_{\mathcal{C}}$, we set $L_{\mathcal{C}} := \sum_{i=1}^{2^d} L_{\mathcal{C}_i}$. If $\sum_{i=1}^{2^d} U_{\mathcal{C}_i} < U_{\mathcal{C}}$, we set $U_{\mathcal{C}} := \sum_{i=1}^{2^d} U_{\mathcal{C}_i}$. After the assignment $L_{\mathcal{C}} \leq n_{\mathcal{C}} \leq U_{\mathcal{C}}$ still holds. We use this technique for all cells considered as heavy (in the order of increasing size), getting better bounds $L_{\mathcal{C}}$ and $U_{\mathcal{C}}$. From these bounds we then compute the values $E_{\mathcal{C}}$ top-down. Since the bounds $L_{\mathcal{C}}$ and $U_{\mathcal{C}}$ are always at least as strong as the bounds of the subcells, we can always easily find integral values $E_{\mathcal{C}}$ satisfying $L_{\mathcal{C}} \leq E_{\mathcal{C}} \leq U_{\mathcal{C}}$ and $\sum_{i=1}^{2^d} E_{\mathcal{C}_i} = E_{\mathcal{C}}$.

**Corollary 5.6.11** *Assume that the statements of Lemma 5.6.8 are true for all grids and all cells.*
*Then for each cell $\mathcal{C}$ identified as heavy we have $(1-4\epsilon)n_{\mathcal{C}} \leq E_{\mathcal{C}} \leq (1+4\epsilon)n_{\mathcal{C}}$.*
*For each cell $\mathcal{C} \in \mathcal{G}_i$ with $i \leq \log_4(\frac{2}{\epsilon\delta})$ we have $E_{\mathcal{C}} = n_{\mathcal{C}}$.*
*All estimates $E_{\mathcal{C}}$ are integral and consistent with the estimates $E_{\mathcal{C}_i}$ for the subcells $\mathcal{C}_i$ of $\mathcal{C}$.*

**Proof :** The proof follows exactly the proof of Corollary 5.6.5. □

We now apply the algorithm described in Section 5.3 to our estimations $E_{\mathcal{C}}$ and compute a coreset.

**Lemma 5.6.12** *If $\delta \leq \frac{\epsilon^{d+2} \cdot Opt}{8 \cdot k \cdot 33^{d+1} \cdot (1+\log n) \cdot d^{(d+2)/2}}$ and $\epsilon < 1/15$, the coreset computed with respect to the values $E_{\mathcal{C}}$ is an $11\epsilon$-coreset of $P$ with probability $1 - \rho$.*

**Proof :** Let $P'$ be a point set that is distributed according to our estimations $E_{\mathcal{C}}$ (so for every useful cell $C$ we have $|P' \cap \mathcal{C}| = E_{\mathcal{C}}$). The proof of Lemma 5.3.5 shows that the coreset computed by our algorithm is an $\epsilon$-coreset for $P'$. Let $Q = \{q_1, \ldots, q_m\}$ be the computed coreset points. Following exactly the proof of Lemma 5.6.6 we can show that if we would know the point sets $P$ and $P'$, we could (using our coreset method) compute mappings $\gamma : P \to Q$ and $\gamma' : P' \to Q$ and corresponding weight functions $w : Q \to \mathbb{N}$ and $w' : Q \to \mathbb{N}$, such that $(Q, w)$ is an $\epsilon$-coreset for $P$ and $(Q, w')$ is an $\epsilon$-coreset for $P'$ and for all $q_i \in Q$ we have:

$$(1 - 5\epsilon)w(q_i) \leq w'(q_i) \leq (1 + 5\epsilon)w(q_i) \tag{5.5}$$

Let $A$ denote the coreset computed by our sample algorithm. From Lemma 5.3.5 we know for each set of centers $C$:

$$\mathit{Means}(A, C) \in (1 \pm \epsilon) \cdot \mathit{Means}(P', C) \ .$$

From the arguments above we know that

$$\mathit{Means}(P', C) \in \frac{1}{1 \pm \epsilon} \cdot \mathit{Means}((Q, w'), C) \subset (1 \pm 2\epsilon) \cdot \mathit{Means}((Q, w'), C) \ .$$

Since the weights $w'(q)$ and $w(q)$ of each coreset-point in $q \in Q$ differ by at most $5\epsilon \cdot w(q)$, we can conclude:

$$\mathit{Means}((Q, w'), C) \in (1 \pm 5\epsilon) \cdot \mathit{Means}((Q, w), C) \ .$$

Since $(Q, \gamma)$ is an $\epsilon$-coreset for $P$, we obtain:

$$\mathit{Means}((Q, w), C) \in (1 \pm \epsilon) \cdot \mathit{Means}(P, C) \ .$$

Alltogether we get for $\epsilon < 1/15$ :

$$\mathit{Means}(A, C) \in (1 \pm \epsilon)^2 \cdot (1 \pm 2\epsilon) \cdot (1 \pm 5\epsilon) \cdot \mathit{Means}(P, C) \subset (1 \pm 11\epsilon) \cdot \mathit{Means}(P, C) \ .$$

□

### 5.6.3 Oblivious Optimization Problems

We again consider $Z$ grids $\mathcal{G}_0, \ldots, \mathcal{G}_{Z-1}$ for $Z = \left\lceil \log\left(\frac{(10\ell)^{d+1} \cdot n(1+\log n) \cdot d^{(d+1)/2} \cdot \tilde{\Delta}}{\epsilon^{d+1} \cdot \lambda^d}\right)\right\rceil + 1$, grid $\mathcal{G}_i$ having cell side length $\frac{1}{2^i}$. In each grid $\mathcal{G}_i$ we pick a random sample $S_i$ of points. To select our random sample we take every point with probability

$$p_i = \min\left\{\frac{\alpha}{\delta \cdot 2^i}, 1\right\}$$

into our sample $S_i$, where

$$\alpha = 6 \cdot \epsilon^{-2} \ln(2 \cdot Z \cdot 2^{Z \cdot d}/\rho)$$

and $\rho$ is the desired error probability of our algorithm. The sampling is done at least $\alpha$-wise independently, which means that for each set $A \subset P$ of at most $\alpha$ points and each partition $\{B, C\}$ of $A$:

$$\mathbf{Pr}[B \subset S_i \wedge C \cap S_i = \emptyset] = (p_i)^{|B|} \cdot (1 - p_i)^{|C|} \ .$$

Essentially this means that for each subset $A \subset P$ of size $\alpha$ the sampling is done independently.

We will show that it follows from a variant of Chernoff bounds [113] that we can approximate the number of points in every heavy cell up to a multiplicative error of $(1 \pm \epsilon)$ just using our point samples. The approximations will furthermore be good enough to detect the heavy cells and to construct a coreset in the same way as described before.

**Definition 5.6.13 (Considered as Heavy)** *For each cell $\mathcal{C}$ in grid $\mathcal{G}_i$ we define $n_{\mathcal{C}}$ as the number of points in $\mathcal{C}$. We define our estimation on the number of points as*

$$\widetilde{n_{\mathcal{C}}} := |S_i \cap \mathcal{C}| \cdot \frac{1}{p_i} \ .$$

*A cell $\mathcal{C}$ in grid $\mathcal{G}_i$ is* considered as heavy*, if*

$$\widetilde{n_{\mathcal{C}}} \geq (1 - \epsilon) \cdot \delta \cdot 2^i \ .$$

**Lemma 5.6.14** *The following events hold with probability at least $1 - \rho/2$ for all grids $\mathcal{G}_i$ and each grid cell in $\mathcal{G}_i$:*

- *If $i \leq \log(\frac{2}{\epsilon\delta})$, then $\widetilde{n_{\mathcal{C}}} = n_{\mathcal{C}}$.*
- *If $\mathcal{C}$ contains at least $\delta \cdot 2^{i-1}$ points, then $(1 - \epsilon) \cdot n_{\mathcal{C}} \leq \widetilde{n_{\mathcal{C}}} \leq (1 + \epsilon) \cdot n_{\mathcal{C}}$.*
- *If $\mathcal{C}$ contains less than $\delta \cdot 2^{i-1}$ points, then $\widetilde{n_{\mathcal{C}}} < (1 - \epsilon) \cdot \delta \cdot 2^i$ (and the cell $\mathcal{C}$ is not considered as heavy).*

**Proof :** The proof is exactly the same as the proof of Lemma 5.6.2. □

If $\widetilde{n_{\mathcal{C}}} \geq (1 - \epsilon) \cdot \delta \cdot 2^i$, a cell $\mathcal{C} \in \mathcal{G}_i$ is considered as heavy. This way, we detect every heavy cell but we also consider some light cells as heavy.

We then compute a coreset by introducing a coreset point in each cell considered as heavy (as described in Section 5.4.1). This will increase the size of our coreset. The following corollaries show that the size of the coreset is still logarithmic in $n$.

**Corollary 5.6.15** *Assume that the statements of Lemma 5.6.14 hold for all cells in all grids.*

*If $\delta \geq \frac{\epsilon^{d+1} \cdot \lambda^d \cdot Opt}{2 \cdot (10\ell)^{d+1} \cdot (1 + \log n) \cdot d^{(d+1)/2}}$, the size of the computed coreset is at most $\frac{9 \cdot (10\ell)^{d+1} \cdot (2 + \log n) \cdot d^{d/2}}{\lambda^{d+1} \cdot \epsilon^{d+1}} = O(\log n / \epsilon^{d+1})$.*

**Proof :** We can easily modify the proof of Lemma 5.4.7 by plugging in $\delta/2$ for the old value of $\delta$. The proof stays exactly the same and we can conclude that the size of the coreset

is at most $\frac{4\cdot Opt}{\delta\cdot\lambda\cdot\sqrt{d}} + (\log n + 2)\cdot 3^d\cdot d^{d/2}$, which is smaller than the stated coreset size for $\delta \geq \frac{\epsilon^{d+1}\cdot\lambda^d\cdot Opt}{2\cdot(10\ell)^{d+1}\cdot(1+\log n)\cdot d^{(d+1)/2}}$.

□

An important property of our sample technique is that although the sample can be large, it just occupies a small number of cells (and can, as we show later, be stored efficiently).

**Lemma 5.6.16** *Let* $\delta \geq \frac{\epsilon^{d+1}\cdot\lambda^d\cdot Opt}{2\cdot(10\ell)^{d+1}\cdot(1+\log n)\cdot d^{(d+1)/2}}$. *Then we have points from at most*

$$\frac{49\cdot Z\cdot(10\ell)^{d+1}\cdot(1+\log n)\cdot d^{d/2}\cdot\ln(2\cdot Z\cdot 2^{Zd}/\rho)}{\epsilon^{d+3}\cdot\lambda^{d+1}} = \widetilde{O}(\log n\cdot\log\widetilde{\Delta}\cdot\log(\rho^{-1})/\epsilon^{d+3})$$

*cells in the union of our sample sets with probability at least* $1-\rho/2$.

**Proof :** Let $\mathcal{G}_i$ be a fixed grid. We determine an upper bound on the number of points in non-center grid cells. Let us recall from the proof of Lemma 5.4.7 that every point except for those contained in the $3^d\cdot d^{d/2}$ center cells has a distance of at least $\frac{\sqrt{d}}{2^{i+1}}$ to the center of gravity and contributes at least $\lambda\cdot\frac{\sqrt{d}}{2^{i+1}}$ to the cost of an optimal solution. Thus the overall number of points in non-center cells is at most $\frac{Opt\cdot 2^{i+1}}{\lambda\cdot\sqrt{d}}$. Let $X_p$ denote the indicator random variable for the event that $p\in S_i$. Let $\mathcal{D}$ denote the set of non-center grid cells. We have $\mathbf{E}[\sum_{p\in\mathcal{D}} X_p] \leq p_i\cdot\frac{Opt\cdot 2^{i+1}}{\lambda\cdot\sqrt{d}} \leq \frac{2\cdot\alpha\cdot Opt}{\delta\cdot\lambda\cdot\sqrt{d}}$. We will assume $\mathbf{E}[\sum_{p\in\mathcal{D}} X_p] = \frac{2\cdot\alpha\cdot Opt}{\delta\cdot\lambda\cdot\sqrt{d}}$ as the distribution of $\sum_{p\in\mathcal{D}} X_p$ displaces towards lower values when $\mathbf{E}[\sum_{p\in\mathcal{D}} X_p] < \frac{2\cdot\alpha\cdot Opt}{\delta\cdot\lambda\cdot\sqrt{d}}$.

Applying Theorem 1 of Section 2.4 we get

$$\mathbf{Pr}\Big[\sum_{p\in\mathcal{D}} X_p \geq \frac{4\cdot\alpha\cdot Opt}{\delta\cdot\lambda\cdot\sqrt{d}}\Big] \leq \mathbf{Pr}\Big[\Big|\sum_{p\in\mathcal{D}} X_p - \mathbf{E}[\sum_{p\in\mathcal{D}} X_p]\Big| \geq \mathbf{E}[\sum_{p\in\mathcal{D}} X_p]\Big]$$
$$\leq e^{-\min\{\lfloor\alpha/2\rfloor,\lfloor\epsilon^2\frac{\alpha}{\delta}/3\rfloor\}} \leq \frac{\rho}{2Z}\ .$$

Therefore, with probability at least $1-\rho/(2Z)$ we have at most $\frac{4\cdot\alpha\cdot Opt}{\delta\cdot\lambda\cdot\sqrt{d}}$ points from non-center cells in our sample. In the worst case no two of these points are contained in the same grid cell and we get an upper bound of $\frac{4\cdot\alpha\cdot Opt}{\delta\cdot\lambda\cdot\sqrt{d}}$ on the number of non-center cells that contain a sample point. Since there are at most $3^d\cdot d^{d/2}$ center cells the number of cells occupied by sample points is at most $\frac{4\cdot\alpha\cdot Opt}{\delta\cdot\lambda\cdot\sqrt{d}} + 3^d\cdot d^{d/2} \leq \frac{49\cdot(10\ell)^{d+1}\cdot(1+\log n)\cdot d^{d/2}\cdot\ln(2\cdot Z\cdot 2^{Zd}/\rho)}{\epsilon^{d+3}\cdot\lambda^{d+1}}$.

Since these arguments hold for each of the $Z$ grids with probability $1-\rho/(2Z)$, the Lemma follows from the union bound. □

To obtain a coreset we use the estimations $\widetilde{n_C}$ of the number of points in heavy cells to identify the cells we consider as heavy (all cells having $\widetilde{n_C} \geq (1-\epsilon)\delta\, 2^i$). Since all heavy cells are considered as heavy we obtain a finer coreset than before. We will now show how to find a good assignment of weights to the computed coreset points, such that the computed coreset is an $\epsilon$-coreset for $P$.

Since the weight of a coreset point will also depend on the number of points in some light cells, we have to estimate the number of points in these cells. To get an estimate for all required cells we use the following procedure. We require that the estimate $\widetilde{n_{\mathcal{C}}}$ for the number of points in a heavy cell is a $(1 \pm \epsilon)$-approximation and that in every cell $\mathcal{C} \in \mathcal{G}_i$ considered as light there are not more than $\delta \cdot 2^i$ points (our coreset construction uses only these assumptions, and they hold according to Lemma 5.6.14 with probability $1 - \rho/2$).

We use exactly the same technique described for k-median to obtain consistent estimates $E_C$:

We call a cell *useful*, if it is either considered as heavy or a direct subcell of a cell considered as heavy. We have to deal with the fact that the sum of the total estimated number of points $\sum_{\mathcal{C}_i \text{ subcell of } \mathcal{C}} \widetilde{n_{\mathcal{C}_i}}$ in the subcells of $\mathcal{C}$ can exceed the estimated number of points $\widetilde{n_{\mathcal{C}}}$ in $\mathcal{C}$. To avoid this we have to compute new integral estimates $E_{\mathcal{C}}$ for the number of points in each *useful* cell $\mathcal{C}$, which still have the guarantee to be near the real value $n_{\mathcal{C}}$ and which are consistent with the values $E_{\mathcal{C}_i}$ of the subcells of $E_{\mathcal{C}}$. We do this by first computing upper and lower bounds $U_{\mathcal{C}}$ resp. $L_{\mathcal{C}}$ on $n_{\mathcal{C}}$ for all useful cells. We will then adjust these bounds to be consistent with the bounds for the subcells. Finally we will use the bounds to compute new estimates $E_{\mathcal{C}}$.

For $i > \log(\frac{2}{\epsilon\delta})$ and every cell $C \in \mathcal{G}_i$ considered as heavy we define $L_{\mathcal{C}} = \lceil \widetilde{n_{\mathcal{C}}}/(1+\epsilon) \rceil$ and $U_{\mathcal{C}} = \lfloor \widetilde{n_{\mathcal{C}}}/(1-\epsilon) \rfloor$. For $i > \log(\frac{2}{\epsilon\delta})$ and every cell $\mathcal{C} \in \mathcal{G}_i$ considered as light we define $L_{\mathcal{C}} = 0$ and $U_{\mathcal{C}} = \lfloor \delta 2^i \rfloor$. For $i \leq \log(\frac{2}{\epsilon\delta})$ and every cell $C \in \mathcal{G}_i$ we define $L_{\mathcal{C}} = \widetilde{n_{\mathcal{C}}}$ and $U_{\mathcal{C}} = \widetilde{n_{\mathcal{C}}}$ (since we know the number of points in $\mathcal{C}$ exactly). Using these definitions we know for every cell that $L_{\mathcal{C}} \leq n_{\mathcal{C}} \leq U_{\mathcal{C}}$).

The estimates $E_{\mathcal{C}}$ can be computed bottom-up by adjusting the bounds $L_{\mathcal{C}}$ and $U_{\mathcal{C}}$ in cases of conflicts:

We first compute new lower and upper bounds $L_{\mathcal{C}}$ and $U_{\mathcal{C}}$ for all useful cells bottom-up. We look at the smallest cell $\mathcal{C}$ considered as heavy. Let $\mathcal{C}_i, i \in \{1, ..., 2^d\}$ be its subcells. If $\sum_{i=1}^{2^d} L_{\mathcal{C}_i} > L_{\mathcal{C}}$, we set $L_{\mathcal{C}} := \sum_{i=1}^{2^d} L_{\mathcal{C}_i}$. If $\sum_{i=1}^{2^d} U_{\mathcal{C}_i} < U_{\mathcal{C}}$, we set $U_{\mathcal{C}} := \sum_{i=1}^{2^d} U_{\mathcal{C}_i}$. After the assignment $L_{\mathcal{C}} \leq n_{\mathcal{C}} \leq U_{\mathcal{C}}$ still holds. We use this technique for all cells considered as heavy (in the order of increasing size), getting better bounds $L_{\mathcal{C}}$ and $U_{\mathcal{C}}$. From these bounds we then compute the values $E_{\mathcal{C}}$ top-down. Since the bounds $L_{\mathcal{C}}$ and $U_{\mathcal{C}}$ are always at least as strong as the bounds of the subcells, we can always easily find integral values $E_{\mathcal{C}}$ satisfying $L_{\mathcal{C}} \leq E_{\mathcal{C}} \leq U_{\mathcal{C}}$ and $\sum_{i=1}^{2^d} E_{\mathcal{C}_i} = E_{\mathcal{C}}$.

**Corollary 5.6.17** *Assume that the statements of Lemma* 5.6.14 *are true for all grids and all cells.*
*Then for each cell $\mathcal{C}$ identified as heavy we have $(1-4\epsilon)n_{\mathcal{C}} \leq E_{\mathcal{C}} \leq (1+4\epsilon)n_{\mathcal{C}}$.*
*For each cell $\mathcal{C} \in \mathcal{G}_i$ with $i \leq \log(\frac{2}{\epsilon\delta})$ we have $E_{\mathcal{C}} = n_{\mathcal{C}}$.*
*All estimates $E_{\mathcal{C}}$ are integral and consistent with the estimates $E_{\mathcal{C}_i}$ for the subcells $\mathcal{C}_i$ of $\mathcal{C}$.*

**Proof :** Exactly as the proof of Corollary 5.6.5 □

We apply the algorithm described in Section 5.4 to our estimations $E_{\mathcal{C}}$ and compute a coreset.

**Lemma 5.6.18** *If $\delta \leq \frac{\epsilon^{d+1} \cdot \lambda^d \cdot Opt}{(10\ell)^{d+1} \cdot (1+\log n) \cdot d^{(d+1)/2}}$ and $\epsilon < 1/12$, the coreset computed with respect to the values $E_{\mathcal{C}}$ is a $67\frac{\ell}{\lambda}\epsilon$-coreset of $P$ with probability $1-\rho$.*

**Proof :** Let $P'$ be a point set that is distributed according to our estimations $E_{\mathcal{C}}$ (so for every useful cell $C$ we have $|P' \cap \mathcal{C}| = E_{\mathcal{C}}$). The proof of Lemma 5.4.4 shows that the coreset computed by our algorithm is an $\epsilon$-coreset for $P'$. Let $Q = \{q_1, \ldots, q_m\}$ be the computed coreset points. Exactly as in the proof of Lemma 5.6.6 we can show that if we would know the point sets $P$ and $P'$, we could (using our coreset method) compute mappings $\gamma : P \to Q$ and $\gamma' : P' \to Q$ and corresponding weight functions $w : Q \to \mathbb{N}$ and $w' : Q \to \mathbb{N}$, such that $(Q, w)$ is an $\epsilon$-coreset for $P$ and $(Q, w')$ is an $\epsilon$-coreset for $P'$ and for all $q_i \in Q$ we have:

$$(1 - 5\epsilon)w(q) \leq w'(q) \leq (1 + 5\epsilon)w(q) \ . \tag{5.6}$$

This guarantees that each solution on $(Q, w)$ can differ from a solution on $(Q, w')$ by at most $10\frac{\ell}{\lambda}\epsilon \cdot \mathit{Opt}(Q, w)$:

Consider a movement of points from $(Q, w)$ to $(Q, w')$. The total movement increases when we first move each moved point to the center of gravity $\mu$ and then in a second step to its destination. Since $w'(q) \geq (1 - 5\epsilon)w(q)$, we know that for each $q \in Q$ at most $5\epsilon w(q)$ points are moved away from $q$. The total movement of the first step therefore is smaller than

$$\sum_{q \in Q} 5\epsilon w(q) d(q, \mu) \ .$$

Since the considered problem is $\ell$-Lipschitz, this movement changes the objective function by at most

$$\ell \cdot \sum_{q \in Q} 5\epsilon w(q) d(q, \mu) \ .$$

Now we look at the second movement step. Since $w'(q) \leq (1 + 5\epsilon)w(q)$ we know that for each $q \in Q$ at most $5\epsilon w(q)$ points are moved from the center of gravity to $q$. The total movement of the second step must therefore be smaller than

$$\sum_{q \in Q} 5\epsilon w(q) d(q, \mu)$$

which changes the objective function again by at most

$$\ell \cdot \sum_{q \in Q} 5\epsilon w(q) d(q, \mu) \ .$$

Since the problem is $\lambda$-mean preserving we conclude that the total change of the objective function caused by both movements is at most

$$10\epsilon\ell \cdot \sum_{q \in Q} w(q) d(q, \mu) \leq \frac{10\ell}{\lambda} \cdot \epsilon \cdot \mathit{Opt}(Q, w) \tag{5.7}$$

We constructed different mappings proving coreset properties for different sets. We use the notation $E : Q \to \mathbb{N}$ for the coreset weight function computed by our sampling algorithm and $m : \gamma(P) \to \gamma'(P)$ for the mapping which moves points as shown in the last paragraph.

We know the following facts:

1. $\gamma : P \to Q$ proves that $(Q, w)$ is a coreset for $P$
2. $m : \gamma(P) \to \gamma'(P)$ changes the cost of a solution by at most $10\frac{\ell}{\lambda}\epsilon \cdot Opt(\gamma(P))$
3. $\gamma' : P' \to Q$ proves that $(Q, w')$ is a coreset for $P'$
4. $\alpha : P' \to Q$ (as constructed by our algorithm) proves that $(Q, E)$ is a coreset for $P'$.

Let $P = (p_1, \ldots, p_n)$ be the set of input points and $s \in S_\Pi(n)$ a solution of the oblivious optimization problem Pi. The mapping $\gamma \circ m \circ \gamma'^{-1} \circ \alpha : P \to \alpha(\gamma'^{-1}(m(\gamma(P))))$ proves that $(Q, E)$ is a $67\frac{\ell}{\lambda}\epsilon$-coreset for $P$:

We write $cost(P)$ for $cost_\Pi^{(n,s)}(P)$ and $Opt(P)$ for $Opt_\Pi(P)$ and conclude:

$$
\begin{aligned}
& cost(\alpha(\gamma'^{-1}(m(\gamma(P))))) && (5.8)\\
\in\ & cost(\gamma'^{-1}(m(\gamma(P)))) \pm \epsilon \cdot Opt(\gamma'^{-1}(m(\gamma(P))))) && (5.9)\\
\subset\ & cost(m(\gamma(P))) \pm 2\epsilon \cdot Opt(\gamma'^{-1}(m(\gamma(P))))) && (5.10)\\
\subset\ & cost(m(\gamma(P))) \pm 4\epsilon \cdot Opt(m(\gamma(P))) && (5.11)\\
\subset\ & cost(\gamma(P)) \pm \left(4\epsilon \cdot Opt(\gamma(P)) + 4\epsilon \cdot 10\frac{\ell}{\lambda}\epsilon \cdot Opt(\gamma(P)) + 10\frac{\ell}{\lambda}\epsilon \cdot Opt(\gamma(P))\right) && (5.12)\\
\subset\ & cost(\gamma(P)) \pm 44\frac{\ell}{\lambda}\epsilon \cdot Opt(\gamma(P)) && (5.13)\\
\subset\ & cost(P) \pm \left(44\frac{\ell}{\lambda}\epsilon \cdot Opt(P) + 44\frac{\ell}{\lambda}\epsilon \cdot \epsilon \cdot Opt(\gamma(P)) + \epsilon \cdot Opt(P)\right) && (5.14)\\
\subset\ & cost(P) \pm 67\frac{\ell}{\lambda}\epsilon \cdot Opt(P) && (5.15)
\end{aligned}
$$

where (5.9) comes from the fact 4, (5.10) from fact 3, (5.11) because of Lemma 5.1.13 and fact 3, (5.12) because of the bounded movement costs of $m$ (fact 2), (5.13) because of $\ell \geq 1$, $\lambda \leq 1$, and $\epsilon \leq 1/2$, and (5.14) because of fact 1. □

# 6 Coresets in Data Streams

In this chapter we introduce algorithms to compute coresets on dynamic geometric data streams. We will combine the coreset results of Chapter 5 with methods of Chapter 3 to sample items in data streams.

We shortly recapitulate the results of Chapter 5, particularly Section 5.6. To state the results for all problems at once we use the following definitions:

For the k-median problem we set
$\delta_0 := \frac{\epsilon^{d+1} \cdot Opt}{4 \cdot k \cdot 10^d \cdot (1+\log n) \cdot d^{(d+1)/2}}$,
$j_0 := \lceil \log(n \cdot \widetilde{\Delta} \cdot \sqrt{d}) \rceil$,
$Z := \left\lceil \log\left( \frac{4 \cdot k \cdot 10^d \cdot n(1+\log n) \cdot d^{(d+1)/2} \cdot \widetilde{\Delta}}{\epsilon^{d+1}} \right) \right\rceil + 1$,
$S_C := \frac{33 \cdot k \cdot 10^d \cdot (2+\log n) \cdot d^{d/2}}{\epsilon^{d+1}}$,
$p_i := \min\left\{ \frac{\alpha}{\delta \cdot 2^i}, 1 \right\}$,
$A := \frac{193 \cdot Z \cdot k \cdot 10^d \cdot (1+\log n) \cdot d^{d/2} \cdot \ln(2 \cdot Z \cdot 2^{Zd}/\rho)}{\epsilon^{d+3}}$, and
$\widetilde{\Delta} := \Delta$.

For the k-means problem we set
$\delta_0 := \frac{\epsilon^{d+2} \cdot Opt}{8 \cdot k \cdot 33^{d+1} \cdot (1+\log n) \cdot d^{(d+2)/2}}$,
$j_0 := \lceil \log(n \cdot \widetilde{\Delta} \cdot \sqrt{d}) \rceil$,
$Z := \left\lceil \log\left( \frac{8 \cdot k \cdot 33^{d+1} \cdot n(1+\log n) \cdot d^{(d+2)/2} \cdot \widetilde{\Delta}}{\epsilon^{d+2}} \right) \right\rceil + 1$,
$S_C := \frac{129(1+\log n) \cdot k \cdot 33^{d+1} \cdot d^{d/2}}{\epsilon^{d+2}}$,
$p_i := \min\left\{ \frac{\alpha}{\delta \cdot 4^i}, 1 \right\}$,
$A := \frac{769 \cdot Z \cdot k \cdot 33^{d+1} \cdot (1+\log n) \cdot d^{d/2} \cdot \ln(2 \cdot Z \cdot 2^{Zd}/\rho)}{\epsilon^{d+4}}$, and
$\widetilde{\Delta} := \Delta^2$.

For oblivious optimization problems we set
$\delta_0 := \frac{\epsilon^{d+1} \cdot \lambda^d \cdot Opt}{(10\ell)^{d+1} \cdot (1+\log n) \cdot d^{(d+1)/2}}$,
$j_0 := \lceil \log(n \cdot \widetilde{\Delta} \cdot \sqrt{d} \cdot \ell) \rceil$,
$Z := \left\lceil \log\left( \frac{(10\ell)^{d+1} \cdot n(1+\log n) \cdot d^{(d+1)/2} \cdot \widetilde{\Delta}}{\epsilon^{d+1} \cdot \lambda^d} \right) \right\rceil + 1$,
$S_C := \frac{9 \cdot (1+\log n) \cdot d^{d/2} \cdot (10\ell)^{d+1}}{\epsilon^{d+1} \cdot \lambda^{d+1}}$,
$p_i := \min\left\{ \frac{\alpha}{\delta \cdot 2^i}, 1 \right\}$,
$A := \frac{49 \cdot Z \cdot (10\ell)^{d+1} \cdot (1+\log n) \cdot d^{d/2} \cdot \ln(2 \cdot Z \cdot 2^{Zd}/\rho)}{\epsilon^{d+3} \cdot \lambda^{d+1}}$, and

$\widetilde{\Delta} := \Delta/\lambda$.

For all problems we set
$\alpha := 6 \cdot \epsilon^{-2} \ln(2 \cdot Z \cdot 2^{Z \cdot d}/\rho)$

and
$\rho := \psi/(3j_0)$,

where $\psi$ is the maximum error probability that our data streaming algorithm is allowed to have.

In Section 5.6 we showed that for a set of $n$ points in $[0,1]^d$ with $Opt \geq 1/\widetilde{\Delta}$ an $\epsilon$-coreset for $k$-median, $k$-means and oblivious optimization problems can be extracted using only the following statistics:

- $Z$ nested grids $\mathcal{G}_0, \ldots, \mathcal{G}_Z$. Each cell in $\mathcal{G}_i$ has side length $\frac{1}{2^i}$.
- In each grid $\mathcal{G}_i$ a sample $S_i$ of points, each point chosen to be in $S_i$ with probability $p_i$. The sampling is assumed to be $\alpha$-wise independently.

Note that we don't need to know the exact location of each sample point. During our construction we only use the information about the number of sample points in certain cells of the grid. Therefore the information we have to compute in our data stream is the following:

1. The coordinates and size of each cell $\mathcal{C}$ occupied by at least one sample point.
2. The number of sample points $S_i \cap \mathcal{C}$ in each occupied cell $\mathcal{C}$.

Notice that according to Lemmas 5.6.4, 5.6.10, and 5.6.16 all information we need can be stored in small space:

**Lemma 6.0.19** *Let $\delta \geq \delta_0/2$. Then we have points from at most $A$ cells in the union of our sample sets with probability at least $1 - \rho/2$.* □

The extracted coreset is a coreset according to Lemmas 5.6.6, 5.6.12, and 5.6.18 and the size of it is small according to Corollarys 5.6.3, 5.6.9, and 5.6.15:

**Lemma 6.0.20** *The following statements hold with probability at least $1 - \rho/2$:*
*If $\delta \leq \delta_0$, we can compute a coreset from only the statistics 1 and 2. The coreset then is an $O(\epsilon)$-coreset of $P$.*
*If $\delta \geq \delta_0/2$, the size of the computed coreset is at most $S_C$.* □

By dividing all point coordinates by $1/\Delta$, we can ensure that all points lie in $[0,1]^d$. We can also assume that we always have $Opt \geq 1/\widetilde{\Delta}$:

For $k$-median we know the exact solution when we currently only have $k$ points in the data stream. We can use a data structure of Lemma 3.5.4 to obtain the exact point coordinates in that

case. Otherwise $Opt \geq 1/\widetilde{\Delta}$, because we have two points of (scaled) distance at least $1/\Delta = 1/\widetilde{\Delta}$ in one cluster.

For k-means we also know the exact solution when we only have $k$ points in the stream. Otherwise $Opt \geq 1/\Delta^2 = 1/\widetilde{\Delta}$, because we have two points of (scaled) distance at least $1/\Delta$ in one cluster.

Having an oblivious optimization problem $\Pi$ we also are able to recover the point set, if it consists of only one point by Lemma 3.5.4. If $P$ consists of at least two points $p$ and $q$, they must have a scaled distance of at least $1/\Delta$. Therefore the total distance of all points to the center of gravity $\mu$ is at least $1/\Delta$. Since $\Pi$ is $\lambda$-mean preserving, we have $Opt \geq \lambda \sum_{p \in P} d(p, \mu) \geq \lambda/\Delta = 1/\widetilde{\Delta}$.

## 6.1 Insertions

We show how to maintain the statistics 1 and 2 when the data stream consists only of insertions of points. Since we do not know the right value of $\delta$ in advance, we follow the approach of the Sections 5.2.3, 5.3.3, and 5.4.3. We start in parallel $j_0$ different instances $I_1, \ldots, I_{j_0}$ of our algorithm, instance $I_j$ with a value of $\delta = \delta(j) = c \cdot 2^j$ for a constant $c$ (see Sections 5.2.3, 5.3.3, and 5.4.3 for the definition of $c$).

For each $j$ we maintain at most $A$ cells $\mathcal{C}_{j,1}, \ldots, \mathcal{C}_{j,A}$ and the number of sample points $s_{j,l}$ within each cell $\mathcal{C}_{j,l}$. For each instance $I_j$ we maintain a value $\text{RUNNING}_j$, which is 1 if the instance is still running or 0 if the instance has been stopped, and a value $c_j$ which denotes the number of cells currently stored by the $j$-th instance.

An insert operation of a point $p$ is done as follows:

```
INSERT(p)
  for each j do
    if RUNNING_j = 1 then
      for each i ∈ {1, ..., Z} do
        Do a biased coin flip with Pr[coin shows heads] = p_i.
        if coin shows heads do
          Let C ∈ G_i denote the cell in grid G_i that contains the point p.
          if ∃_{l∈{1,...,c_j}} C_{j,l} = C do
            set s_{j,l} ← s_{j,l} + 1. // update number of sample points within this cell
          else do
            if c_j > A do
              set RUNNING_j = 0. // stop storing cells for this j
            else do
              set c_j ← c_j + 1.
              set C_{j,c_j} ← C. // store cell occupied by point
              set s_{j,c_j} ← 1. // store number of sample points within this cell
```

The choice of $j_0$ ensures that for one $j$ we have $\frac{1}{2} \cdot \delta_0 \leq \delta(j) \leq \delta_0$ (see Sections 5.2.3, 5.3.3, and 5.4.3). According to Lemma 6.0.19 for this choice of $\delta = \delta(j)$ we have at most $A$ cells occupied by sample points. Therefore we have $c_j \leq A$ during the algorithm and $\text{RUNNING}_j = 1$. From Lemma 6.0.20 we know that when we extract a coreset from the statistics for this value of $j$, we obtain an $O(\epsilon)$-coreset for the respective problem. We also obtain an $O(\epsilon)$-coreset from all statistics having smaller values of $j$, since they used a smaller value of $\delta = \delta(j)$.

We use the smallest value of $j$ such that $\text{RUNNING}_j = 1$. From the corresponding statistics $\mathcal{C}_{j,1}, \ldots, \mathcal{C}_{j,c_j}$ and $s_{j,1}, \ldots, s_{j,c_j}$ we compute our coreset, which is an $O(\epsilon)$-coreset for $P$. If the coreset size is bigger than $S_C$ we know from Lemma 6.0.20 that $\delta(j) < \delta_0/2$. Therefore we can set $j \leftarrow j + 1$, still knowing that $\delta(j) \leq \delta_0$ and the computed coreset for this value of $j$ is also an $O(\epsilon)$-coreset. We go on with this method until the size of the constructed coreset is at most $S_C$.

**Theorem 18** *Given a data stream of point insertions our streaming algorithm maintains a data structure for $\epsilon$-coresets for $k$-median, $k$-means, and oblivious optimization problems. At any point of time an $O(\epsilon)$ coreset for the respective problem can be extracted with probability $1-\psi$.*

*The data structure for $k$-median needs $\widetilde{O}(k \cdot \log^4(\Delta) \cdot \log(1/\psi)/\epsilon^{d+3})$ space, the data structure for $k$-means needs $\widetilde{O}(k \cdot \log^4(\Delta) \cdot \log(1/\psi)/\epsilon^{d+4})$ space and the data structure for oblivious optimization problems needs $\widetilde{O}(\log^4(\Delta) \cdot \log(1/\psi)/\epsilon^{d+3})$ space.*

*An insertion can be processed in $O(\log(\Delta) \cdot \log(k \cdot \Delta/\epsilon))$ time.*

*An $\epsilon$-coreset for $k$-median can be extracted in $O(k \log^3 \Delta/\epsilon^{d+1})$ time.*

*An $\epsilon$-coreset for $k$-means can be extracted in $O(k \log^3 \Delta/\epsilon^{d+2})$ time.*

*An $\epsilon$-coreset for oblivious optimization problems can be extracted in $O(\log^3 \Delta/\epsilon^{d+1})$ time.*

**Proof :** We set $\rho = \psi/(3 \cdot j_0)$. By the union bound Lemma 5.6.2 and Lemma 5.6.4 hold with probability $1 - \psi$ for all choices of $j$. Thus with probability $1 - \psi$ the returned coreset is an $O(\epsilon)$-coreset.

We will now count the number of memory cells we use. For each of the $j_0 = O(\log \Delta)$ instances we maintain at most $A$ counters. They occupy $j_0 \cdot A$ memory cells, each consisting of $O(\log \Delta)$ bits. The respective values of $A$ lead to the stated memory bounds.

When a new point arrives in the stream we process one for-loop for each of the $j_0$ values of $j$ and each of the $Z$ values of $i$. Using hash tables the query $\exists_{l \in \{1,\ldots,c_j\}} \mathcal{C}_{j,l} = \mathcal{C}$ can be done in constant time. All other operations of the algorithm run in constant time as well. Therefore the time to insert a point is $O(Z \cdot j_0) = O(\log(\Delta) \cdot \log(k \cdot \Delta/\epsilon))$.

We will now bound the coreset extraction time for $k$-median. Let us first assume we know the right value of $\delta$ in advance. Since (according to the proof of Lemma 5.2.8) there are at most $O(k \log n/\epsilon^{d+1})$ heavy cells in each grid, we have a bound of $O(k \log^2 \Delta/\epsilon^{d+1})$ on the number of heavy cells. The marking process yielding to the actual coreset can be seen as a quadtree traversal. Since each inner node of this tree corresponds to a heavy cell, the tree traversal can be done in time $O(k \log^2 \Delta/\epsilon^{d+1})$. Since we have to do this process for each value of $\delta$ (and then choose the minimum value with a small coreset), we get a total running time of $O(k \log^3 \Delta/\epsilon^{d+1})$.

For $k$-means resp. oblivious optimization problems the number of heavy cells in each grid is $O(k \log n/\epsilon^{d+2})$ resp. $O(\log n/\epsilon^{d+1})$. Therefore we get a total coreset extraction time of $O(k \log^3 \Delta/\epsilon^{d+2})$ for $k$-means and $O(\log^3 \Delta/\epsilon^{d+1})$ for oblivious optimization problems. □

## 6.2 Deletions

When we allow deletions two problems occur. First, when we encounter a DELETE operation of a point $p$, we have to decide if this point is a sample point. We achieve this by replacing the coin flips in our algorithm by the use of an $\alpha$-wise independent hashfunction $h_{i,j} : \Delta^d \to \left[\left\lceil \frac{\Delta^d}{\rho} \right\rceil\right]$. We choose all points $p$ having $h_{i,j}(p) < p_i \cdot \left\lceil \frac{\Delta^d}{\rho} \right\rceil$ as sample points.

**Lemma 6.2.1** *The choice of the sample points is done $\alpha$-wise independently. Each point $p \in P$ is chosen to be a sample point with probability $p \in [p_i, p_i + \frac{\rho}{\Delta^d}]$. The total statistical difference to the method which samples each point with probability $p_i$ is at most $\rho$.*

**Proof :**

$$p = \frac{\left\lceil p_i \cdot \left\lceil \frac{\Delta^d}{\rho} \right\rceil \right\rceil}{\left\lceil \frac{\Delta^d}{\rho} \right\rceil} \geq p_i$$

and

$$p = \frac{\left\lceil p_i \cdot \left\lceil \frac{\Delta^d}{\rho} \right\rceil \right\rceil}{\left\lceil \frac{\Delta^d}{\rho} \right\rceil} \leq \frac{p_i \cdot \left\lceil \frac{\Delta^d}{\rho} \right\rceil + 1}{\left\lceil \frac{\Delta^d}{\rho} \right\rceil} = p_i + \frac{1}{\left\lceil \frac{\Delta^d}{\rho} \right\rceil} \leq p_i + \frac{\rho}{\Delta^d} \ .$$

Since we have at most $\Delta^d$ points, the errors in the probability to sample each single point sum up to a total statistical difference of at most $\rho$. □

We conclude that for a fixed $j$ our sampling method behaves with probability $1-\rho$ exactly like an $\alpha$-wise independent sampling of each point with probability $p_i$. Since $\rho = \psi/(3j_0)$ we have that with probability $1 - \psi/3$ all samplings behave exactly as demanded.

The second problem is that we cannot stop an instance of our algorithm if the number $c_j$ of occupied cells exceeds $A$. For example, it could happen that in the first half of the stream many points are inserted and the number of occupied cells $c_j$ is way too large. But then most of these points can be deleted in the second half of the stream such that we eventually have less than $A$ occupied cells. If we want to obtain a coreset at that point of time we have to know the respective statistics 1 and 2.

To overcome this problem we use the data structure of Lemma 3.5.4. The data structure supports update operations on the entries of a high dimensional vector $x$ and is able to recover the whole vector, as long as the support of the vector is smaller than $S_C$.

Let $j$ be fixed. Let $\mathcal{C}_0, \ldots, \mathcal{C}_t$ with $t = O(\Delta^d)$ denote the set of all cells in all grids $\mathcal{G}_i$ for one fixed $j$. Let $x_i$ denote the number of sample points in the cell $\mathcal{C}_i$. When we are able to recover the whole vector $x = (x_0, \ldots, x_t)$ after insert- and delete- operations of sample points, we can reconstruct the statistics 1 and 2. Therefore for each $j$ we use a data structure $\text{RECOVER}_j$ of Lemma 3.5.4 with parameters $U = \Theta(\Delta^d)$, $M = \Theta(\Delta^d)$, and error probability parameter $\rho = \psi/(3j_0)$. Then with probability $1 - \psi/3$ the structures $\text{RECOVER}_j$ work for each $j$.

Let $\text{UPDATE}_j$ denote an UPDATE operation on the data structure $\text{RECOVER}_j$. We implement INSERT and DELETE operations of points in the following way:

INSERT(p)
  **for each** j **do**
    **for each** $i \in \{1, \ldots, Z\}$ **do**
      **if** $h_{i,j}(p) < p_i \cdot \left\lceil \frac{\Delta^d}{\rho} \right\rceil$ **do** // p is sample point
        Let $\mathcal{C}_l \in \mathcal{G}_i$ denote the cell that contains the point p.
        UPDATE$_j(l, 1)$.

DELETE(p)
  **for each** j **do**
    **for each** $i \in \{1, \ldots, Z\}$ **do**
      **if** $h_{i,j}(p) < p_i \cdot \left\lceil \frac{\Delta^d}{\rho} \right\rceil$ **do** // p is sample point
        Let $\mathcal{C}_l \in \mathcal{G}_i$ denote the cell that contains the point p.
        UPDATE$_j(l, -1)$.

The method ensures that $x_l$ always represents the number of sample points within the cell $\mathcal{C}_l$.

**Theorem 19** *Given a data stream of point insertions and deletions our streaming algorithm maintains a data structure for $\epsilon$-coresets for k-median, k-means. At any point of time an $O(\epsilon)$-coreset for the respective problem can be extracted with probability $1 - \psi$.*

*The data structure for k-median needs $\widetilde{O}(k \cdot \log^6(\Delta) \cdot \log(\Delta/\psi)/\epsilon^{d+3})$ space, the data structure for k-means needs $\widetilde{O}(k \cdot \log^6(\Delta) \cdot \log(\Delta/\psi)/\epsilon^{d+4})$ space and the data structure for oblivious optimization problems needs $\widetilde{O}(\log^6(\Delta) \cdot \log(\Delta/\psi)/\epsilon^{d+3}))$ space.*

*Insertions and deletions of points can be processed in $\widetilde{O}(k \cdot \log^6(\Delta) \cdot \log(\Delta/\psi)/\epsilon^{d+3})$ time for k-median, for k-means in $\widetilde{O}(k \cdot \log^6(\Delta) \cdot \log(\Delta/\psi)/\epsilon^{d+4})$ time and for oblivious optimization problems in $\widetilde{O}(\log^6(\Delta) \cdot \log(\Delta/\psi)/\epsilon^{d+3}))$ time.*

*An $\epsilon$-coreset for k-median can be extracted in $\widetilde{O}(k \log^5 \Delta \cdot \log(\Delta/\psi)/\epsilon^{d+3})$ time.*

*An $\epsilon$-coreset for k-means can be extracted in $\widetilde{O}(k \log^5 \Delta \cdot \log(\Delta/\psi)/\epsilon^{d+4})$ time.*

*An $\epsilon$-coreset for oblivious optimization problems can in $\widetilde{O}(\log^5 \Delta \cdot \log(\Delta/\psi)/\epsilon^{d+3})$ time be extracted.*

**Proof :** We have that with probability $1 - \psi/3$ the hash functions doing the sampling behave like sampling each point $\alpha$-wise independently with probability $p_i$. We condition on this event.

We used data structures RECOVER$_j$ that work with probability $1 - \psi/3$ for all j. We condition on this event.

In that event we can recover the whole vector x, which represents exactly the statistics 1 and 2. Therefore we can construct an $O(\epsilon)$-coreset. The coreset construction works with probability $1 - \psi/3$ according to Lemmas 5.6.2 and Lemma 5.6.4.

Let us now bound the space needed. For $j_0$ values of j we have to store the data structure RECOVER$_j$ of Lemma 3.5.4, each using space

$$O\left(A \cdot (\log A + \log(1/\rho)) \cdot \log^2 \Delta\right) = \widetilde{O}\left(A \cdot \log^2(\Delta) \cdot \log(\Delta/\psi)\right) \quad .$$

Therefore the total space to store all data structures $\text{RECOVER}_j$ is $\widetilde{O}\left(A \cdot \log^3(\Delta) \cdot \log(\Delta/\psi)\right)$. The space consumption of the hash functions is negligible compared to the space consumption of the $\text{RECOVER}_j$ structures. We obtain the stated memory bounds by inserting the right values of $A$.

To process an update we have to process $j_0 \cdot Z$ update operations on RECOVER structures, each taking $O(A \cdot (\log A + \log(1/\rho)) \cdot \log \Delta)$ time. Alltogether this takes $\widetilde{O}(Z \cdot A \cdot \log^2 \Delta \cdot \log(\Delta/\psi))$ time, which dominates the whole processing time. Plugging in the values of $Z$ and $A$ leads to the stated update times.

To extract the coreset for a fixed value of $j$ we first have to extract the statistics 1 and 2 from $\text{RECOVER}_j$. This takes $O(A \cdot (\log A + \log(\Delta/\psi)) \cdot \log \Delta) = \widetilde{O}(A \cdot \log \Delta \cdot \log(\Delta/\psi))$ time. Plugging in the value $A$ and multiplying that by $j_0 = O(\log \Delta)$ (the number of possible values of $j$), we obtain a total recovery time for k-median of $\widetilde{O}(k \log^5 \Delta \cdot \log(\Delta/\psi)/\epsilon^{d+3})$, which dominates the coreset extraction time of $O(k \log^3 \Delta/\epsilon^{d+1})$. For k-means and oblivious optimization problems we get the stated results in the same way. □

## 6.3 Maximum Spanning Tree

In our publication [43] we stated the problem to find a maximum spanning tree of points in a Euclidean space as an oblivious optimization problem, which is solvable using our coreset method. Although this can be done in general, the problem is not $\ell$-Lipschitz for a constant $\ell$, and therefore the proofs of the previous chapters do not apply. In this section we want to close this gap to the publication [43] and develop a very simple and more effective method to construct a coreset for the maximum spanning tree problem.

We will first define the maximum spanning tree problem as an oblivious optimization problem.

**Definition 6.3.1 (Euclidean MaxSP)** *The* Euclidean maximum spanning tree problem (MaxSP) *asks for a spanning tree connecting all points of a given input point set $P \subset \mathbb{R}^d$ of size $n$, such that the total length of all tree edges is maximized.*

*The problem can be formulated as an oblivious optimization problem:*

*We look at a complete graph $K_n$ with node set $\{1, \ldots, n\}$. A feasible solution $s$ for the MaxST problem is then a spanning tree of $K_n$. Let $T$ be the set of edges of this spanning tree of $K_n$. The cost of $s$ on $P = \{p_1, \ldots, p_n\}$ is then given by*

$$cost^{(n,s)}_{maxtsp}(p_1, \ldots, p_n) = \sum_{(a,b) \in T} d(p_a, p_b) \ .$$

This definition of MaxSP as an oblivious optimization problem also defines $\epsilon$-*coresets* for MaxSP by Definition 5.1.12.

We first show how to construct a coreset when we have access to the whole input point set $P$.

We compute an approximation $\widetilde{B}$ to the biggest extent of the point set in one dimension. Especially we set

$$B := \min_{p,q \in P} \min_{i \in \{1,\ldots,d\}} \left| p^{(i)} - q^{(i)} \right| \ .$$

and assume that we have computed an approximation $\widetilde{B}$ satisfying $B \leq \widetilde{B} \leq 2 \cdot B$.

We then introduce a grid consisting of square cells of side length $\frac{\epsilon \cdot \widetilde{B}}{8\sqrt{d}}$. Notice that at most $(8 \cdot \sqrt{d}/\epsilon)^d$ cells in the grid are occupied by points of $P$.

We introduce a coreset point in each non-empty grid cell and map all input points from $P$ lying in a cell $\mathcal{C}$ to the corresponding coreset point in $\mathcal{C}$.

**Lemma 6.3.2** *The construction given above constructs an $\epsilon$-coreset for $P$. The coreset size is $O(\frac{1}{\epsilon^d})$.*

**Proof :** Let $s$ be an arbitrary solution. We denote by $\mathcal{T}$ the corresponding spanning tree connecting the points from $P$. $\mathcal{T}$ contains less than $n$ edges. Since each endpoint of an edge is moved a distance at most $\epsilon \cdot \widetilde{B}/8$, we conclude that the length of each edge changes by at most $\epsilon \cdot \widetilde{B}/4$. The total change of the cost of the solution $s$ is therefore at most $n \cdot \epsilon \cdot \widetilde{B}/4 \leq n \cdot \epsilon \cdot B/2$.

We now show that this change is smaller than $\epsilon \cdot \mathit{Opt}_{\Pi}(P)$ by constructing a spanning tree having large cost. We first connect the two points $p_1$ and $p_2$ of $P$ defining the extent $B$ by an edge. We iterate over the rest of the nodes. A node $q \in P \setminus \{p_1, p_2\}$ is connected to $p_1$ by an edge iff $d(q, p_1) \geq d(q, p_2)$. Otherwise $q$ is connected to $p_2$ by an edge. The cost of the resulting spanning tree is then at least

$$d(p_1, p_2) + (n-2) \cdot d(p_1, p_2)/2 \geq n \cdot B/2 \ .$$

Therefore the costs $\mathit{Opt}_{\Pi}(P)$ of an optimum solution must be greater than $n \cdot B/2$. It follows that the change of the cost of the solution $s$ by mapping points to coreset points is bounded by

$$n \cdot \epsilon \cdot B/2 \leq \epsilon \cdot \mathit{Opt}_{\Pi}(P) \ .$$

□

The statistics we need to construct a coreset can be easily maintained in a dynamic geometric data stream consisting of insertions and deletions of points from $\{0, \ldots \Delta - 1\}^d$ using techniques of the previous sections. Let $\psi$ be a desired error probability we want to achieve.

We start $\lceil \log(d \cdot \Delta) + 1 \rceil$ parallel instances of a streaming algorithm, each with a different value of $j \in \{0, \ldots, \lceil \log(d \cdot \Delta) \rceil\}$. For each instance we set $\widetilde{B}_j := 2^j$.

Instance $j$ introduces a square grid $\mathcal{G}_j$ of cell side length $\frac{\epsilon \cdot \widetilde{B_j}}{8\sqrt{d}}$. Let $C_{j,1}, \ldots, C_{j,l}$ be the cells of $\mathcal{G}_j$. The instance uses the data structure of Lemma 3.5.4 with values $U = \Delta^d$, $M = \Delta^d$, $A = (8 \cdot \sqrt{d}/\epsilon)^d$, and $\delta = \psi / \lceil \log(d \cdot \Delta) + 1 \rceil$, which is able to recover the support of a large vector $x$ under UPDATE operations on $x$, when the support is smaller than $A$. For each INSERT$(p)$ operation in the stream instance $j$ discovers the cell $C_{j,i}$ containing the point $p$. It then triggers an UPDATE$(i, 1)$ operation on its data structure, increasing the value $x_i$ by 1. A DELETE$(p)$

operation is transformed into an $\text{UPDATE}(i, -1)$ operation on the data structure, decreasing the value $x_i$ by 1. This way we ensure that the vector entry $x_i$ always equals the number of cells in $C_{j,i}$.

We can extract a coreset in the following way. We assume that all data structures to recover the vectors work well (which happens by the Union Bound with probability $1 - \psi$). We query all data structures of all instances to recover the support of their corresponding vectors. The data structures return FAIL only if the support of their corresponding vector is larger than $A$. This way we can decide which grids contain at most $A$ occupied cells.

We take the instance with the lowest value of $j$ having at most $A$ occupied cells and use the data structure of that instance to recover the corresponding vector $x$. This vector contains the information about the set of occupied cells in $\mathcal{G}_j$ and the number of points within each cell. Using that information we construct a coreset by introducing a coreset point within each occupied cell of $\mathcal{G}_j$ and setting the weight of each coreset point to the number of input points in the corresponding cell.

**Theorem 20** *Given a dynamic geometric data stream of insert and delete operations of points from $\{0, \ldots, \Delta - 1\}^d$, there is an algorithm that maintains a data structure for $\epsilon$-coresets for MaxST. At any point of time an $\epsilon$-coreset can be extracted with probability $1 - \psi$. The size of the coreset is $O(1/\epsilon^d)$. The data structure uses $O\left(\frac{1}{\epsilon^d} \cdot \log\left(\frac{\log \Delta}{\epsilon \cdot \psi}\right) \cdot \log^2 \Delta\right)$ space. Insertions and deletions of points can be processed in $O\left(\frac{1}{\epsilon^d} \cdot \log\left(\frac{\log \Delta}{\epsilon \cdot \psi}\right) \cdot \log \Delta\right)$ time.*

**Proof :** We first prove that the returned coreset is indeed an $\epsilon$-coreset for MaxST. We know that there is one instance started with a value of $\widetilde{B_{j_0}}$ satisfying $B \leq \widetilde{B_{j_0}} \leq 2 \cdot B$. By the argumentation above we have at most $A$ occupied cells in the corresponding grid. If we would construct a coreset from the information of that instance, this would be an $\epsilon$-coreset. However, we construct the coreset from the information of the instance having the smallest value of $j$ and at most $A$ occupied grid cells. Therefore the coreset is constructed from an instance $j$ with value $j \leq j_0$. We have either $j = j_0$ or the grid of instance $j$ is even finer than that of instance $j_0$. Therefore the constructed coreset is an $\epsilon$-coreset. It's size is at most $A$.

The space and update time we need is dominated by the space and update time of the $\lceil \log(d \cdot \Delta) + 1 \rceil$ data structures from Lemma 3.5.4. This leads to the stated space requirements and update times.

□

# 7 A Kinetic Data Structure for MaxCut

In this chapter we will focus on clustering moving points as described in the framework of *kinetic data structures* (KDS). The framework of kinetic data structures has been introduced by Basch et al. [12] and it has been used since as the central model of studying geometric objects in motion, see, e.g., [2, 12, 55, 56] and the references therein. The KDSs are data structures for maintaining a certain attribute (for example, in the case of a clustering problem, assignment of the points to the clusters) for a set of continuously moving geometric objects. The main idea underlying the framework of KDSs is that even if the input objects are moving in a continuous fashion, the underlying combinatorial structure of the moving objects changes only at discrete times. Therefore, there is no need to maintain the data structure continuously but rather only when certain combinatorial *events* happen: a KDS maintains a configuration function of interest by watching for updates needed to be performed when an event occurs.

In the kinetic setting, we consider a set of points in $\mathbb{R}^d$ that are continuously moving. Each point follows a (known) trajectory that is defined by a continuous function of time; for simplicity of presentation, we will assume that it is a linear function. In other words, every point is moving with a constant speed along a line; the line and the speed are the parameters of the movement of a given point. Additionally, we allow the points to change their trajectory, i.e., to perform a *flight plan update*.

To measure the quality of a KDS, we will consider the following two most important performance measures (for more details, see, e.g., [55, 56]): the time needed to update the KDS when an event occurs and the bound for the number of events that may occur during the motion. Another important measure is the time to handle flight plan updates.

In this chapter we describe a kinetic data structure to maintain a $(1-\epsilon)$-approximation of a maximum cut. Our data structure supports queries of the type

- *to which side of the partition belongs query point* $p$*?*

To support such a query the data structure maintains a subdivision of the space that has complexity $O(\log n/\epsilon^{d+1})$. Each cell of the subdivision is colored red or blue. Every point located in a red cell is red and every point in a blue cell is blue. Then our colored partition into red and blue points is a $(1-\epsilon)$-approximation to the MaxCut.

Our data structure uses two auxiliary kinetic data structures, kinetic turnament trees as defined in Section 7.1 and a data structure to approximate the bounding cube as defined in Section 7.2.

## 7.1 Kinetic Turnament Trees

In this section we recap a construction of *randomized turnament trees* from Basch [11]. There are other structures leading to better amortized time bounds (e.g. [16]). We will use the construction from [11] because it deterministically achieves almost the same runtime bounds, but these bounds hold even in the worst case.

A randomized turnament tree is a randomized data structure to maintain the maximum (or the minimum, depending on applications) point from a set $P = \{p_1, \ldots, p_n\}$ of $n$ linearly moving points in $\mathbb{R}^1$. It stores all points in the leafs of a binary tree. Inner nodes of the tree always store the bigger point of the two child nodes (and therefore the maximum of the subtree). It also stores an *event queue*, that is a priority queue holding the times of the next events when inner nodes will change (when we get a new maximum in the subtree, i.e. when the two children of a node change their order).

When a point stored in an inner node changes, the maximum of the subtree must have changed. The number of times this can happen is bounded by the number of nodes in the subtree. Summing up over all inner nodes we get that the total number of changes of inner nodes is at most $O(n \cdot \log n)$.

When an inner node changes at an event, it can lead to changes of the inner nodes above. Notice that we treat these additional changes as seperate events. Therefore each such event can be processed in $O(\log n)$ time (we just have to adapt the event queue and test if we have an immidiate event for the parent node as well).

An insertion of a point can be done by inserting a leaf into the tree and adjusting the inner nodes in time $O(\log^2 n)$. A deletion of a point can be done by deleting the respective leaf in time $O(\log^2 n)$, adjusting the inner nodes, and move the rightest leaf to the position of the deleted leaf in time $O(\log^2 n)$ to balance the tree.

**Theorem 21 ([11])** *Let $P$ be an initially empty set of points moving along linear trajectories in $\mathbb{R}^1$. Let $\sigma = \sigma_1, \ldots, \sigma_m$ be a sequence of $m$ operations $\sigma_i$ of the form $\textsc{Insert}(p, t_i)$ and $\textsc{Delete}(p, t_i)$, such that for any two operations $\sigma_i, \sigma_j$ with $i < j$ we have $t_i < t_j$ (the operations are performed sequentially in time). An $\textsc{Insert}(p, t_i)$ inserts at time $t_i$ point $p$ into $P$. A $\textsc{Delete}(p, t_i)$ removes $p$ from $P$ at time $t_i$. A kinetic turnament tree maintains the biggest element of $P$. It requires $O(\log m)$ time to process an event and the expected number of events is $O(m \log m)$. Insertions and deletions are performed in expected time $O(\log^2 m)$.*

## 7.2 Approximating the Bounding Cube

Our data structure uses an data structure from Agarwal, Har-Peled, and Varadarajan [2]:

**Theorem 22 ([2])** *Let $P$ be a set of $n$ points moving in $\mathbb{R}^d$. If $P$ is moving linearly, then after $O(n)$ preprocessing, we can construct a kinetic data structure of size $O(1)$ that maintains a 2-approximation of the smallest orthogonal box containing $P$. The data structure processes $O(1)$ events, and each event takes $O(1)$ time. The sides of the maintained box are moving linearly between the events.*

*It can be decided in constant time if a flight plan update of a point* $p$ *changes the data structure. At each point of time only flight plan updates of a constant number of points can potentially change the data structure.*

We use this data structure to efficiently maintain a bounding cube $B$ of $P$ having the following properties.

- All points lie in $B$ during the whole movement of points.
- There is always one dimension, such that the extent of the point set in the dimension is at least half the side length of $B$.
- The data structure to maintain the bounding cube processes $O(d^2)$ events.
- Between these events all side borders of the bounding cube are moving linearly.
- Given a flight plan update we can decide in time $O(d)$ if the data structure has to be updated.
- At each point of time only flight plan updates of $O(d)$ points can potentially change the data structure.

For each dimension we maintain a 2-approximation of the 1-dimensional extent using the KDS from [2] (see Theorem 22 above). Using these approximations we can easily maintain an approximation $B$ of a smallest bounding cube of $P$ by maintaining a cube having the highest extent as side length.

Each extent data structure processes $O(1)$ events. Between such events the dimension that defines the size of the bounding cube can change although none of the one dimensional extent data structures processes an event. This happens when the size of the bounding cube is determined by a new one dimensional extent. This can happen at most $d$ times. Therefore, we can have at most $O(d^2)$ events.

We store the approximation of each extent in a kinetic turnament tree whose priority is given by the width of the 1-dimensional approximations.

## 7.3 The Kinetic Data Structure for MaxCut

Our kinetic data structure for MaxCut uses the data structure of the last section to maintain a bounding cube $B$. This data structure processes $O(d^2)$ events, which we call *major events*.

Between all major events we will set up a data structure to maintain a coreset under a linear movement of the bounding box $B$. We will use the technique described in Section 5.6.3 to construct coresets for oblivious optimization problems using point samples. Recall that MaxCut (with objective function scaled by $1/n$) is an oblivious optimization problem which is $\ell$-Lipschitz with $\ell = 1$ and $\lambda$-mean preserving with $\lambda = 1/4$.

The coreset technique assumes that all points always lie within $[0,1]^d$ and that we have a lower bound $1/\widetilde{\Delta}$ on *Opt* (recall that *Opt* denotes the cost of an optimum cut devided by $n$). This can easily be achieved in the following way: Between two major events we know that the movement of the bounding cube $B$ is linear. At each major event we scale and move all coordinates in a way such that the bounding cube $B$ is $[0,1]^d$. Since the scaling function is linear in the time $t$, after this scaling still all point movements are linear.

Since there is always one dimension such that the extent of the point set in that dimension is at least half the side length of the bounding cube, the sum of distances of these two points to the center of gravity is at least $1/2$. Since MaxCut is $1/4$-mean preserving, we know that $Opt \geq 1/\widetilde{\Delta}$ for $\widetilde{\Delta} = 8$.

In the following we assume that all points lie in $[0,1]^d$ and that the bounding cube $B = [0,1]$ does not move.

We will maintain $Z$ nested square grids $\mathcal{G}_0, \ldots \mathcal{G}_Z$ for

$$Z = \left\lceil \log \left( \frac{8 \cdot 10^{d+1} \cdot 4^d \cdot n(1+\log n) \cdot d^{(d+1)/2}}{\epsilon^{d+1}} \right) \right\rceil + 1 \ .$$

Grid $\mathcal{G}_i$ partitions $[0,1]^d$ into $2^{id}$ cells of side length $1/2^i$. We assume that these cells are numbered from 1 to $2^{id}$ and that the index of a cell can be computed efficiently from its coordinates and that the neighbors of a cell can also be computed efficiently.

Let $j_0 := \lfloor \log(8 \cdot n \cdot \sqrt{d}) \rfloor$. For $j \in \{0, \ldots, j_0\}$ let be

$$\delta = \delta(j) = \frac{\epsilon^{d+1}}{8 \cdot (1+\log n) \cdot d^{(d+1)/2} \cdot (40)^{d+1}} \cdot 2^j \ .$$

For each value of $j \in \{0, \ldots, j_0\}$ and each grid $\mathcal{G}_i$ we create a sample $S_{i,j}$, each point chosen independently with probability $p_{i,j} = \min\{\frac{\alpha}{\delta \cdot 2^i}, 1\}$, where $\alpha = 6 \cdot \epsilon^{-2} \cdot \ln(2 \cdot Z \cdot 2^{Z \cdot d} \cdot j_0/\psi)$.

To simplify the notation we set

$$K := \frac{6 \cdot \ln(2 \cdot Z \cdot 2^{Z \cdot d} \cdot j_0/\psi) \cdot 8 \cdot (1+\log n) \cdot d^{(d+1)/2} \cdot 40^{d+1}}{\epsilon^{d+3}} = \frac{\alpha \cdot 2^j}{\delta} \ .$$

Notice that $K$ is polylogarithmic in $n$ and does not depend on $i$ or $j$, and that

$$p_{i,j} = \min\left\{ \frac{K}{2^{i+j}}, 1 \right\}$$

Following the ideas of Section 5.4.3 there is at least one value of $\delta(j)$, such that the coreset constructed using this value of $\delta = \delta(j)$ is smaller than a constant $S = O(\log n/\epsilon^{d+1})$ (Corollary 5.6.15), and an $O(\epsilon)$-coreset for MaxCut (Lemma 5.6.18). We can easily identify such a value of $\delta$ by taking the coreset constructed with the smallest value of $\delta$, such that the size is at most $S$.

For each $1 \leq i \leq Z$, we maintain under the movement of points

- the set of all cells containing sample points, and
- the number of sample points in each non-empty cell.

A $(1+\epsilon)$-approximate solution on the coreset can be efficiently computed using the algorithm of Section 5.5.3 in time $O\left(\log^2 n \cdot 2^{((1/\epsilon)^{O(1)})}\right)$.

**The data structure.** At each major event we calculate a new linear scaling of the movements of points, such that all points lie in the scaled bounding cube $B = [0,1]^d$ as described earlier. We will then setup the following data structure and maintain it until the next major event triggers.

We assume that the cells in grid $\mathcal{G}_i$ are numbered from 1 to $2^{id}$. For each sample set $S_{i,j}$ we maintain a search tree $T_{i,j}$ that stores the cells in grid $\mathcal{G}_i$ that contain at least one point from $S_{i,j}$. For each non-empty cell we maintain $2d$ kinetic turnament trees. For $1 \leq k \leq d$ we maintain one kinetic max-turnament tree and one kinetic min-turnament tree, where the priority of points is given by their $k$-th coordinate. To implement these turnament trees we use a kinetic turnament trees that efficiently supports insertion and deletions of points (see Section 7.1).

**The events.** Additionally to the events caused by the kinetic turnament trees, our data structure stores the following (possible) events in a global event queue. For each grid $\mathcal{G}_i$ and each non-empty cell we have an event for each dimension $k$, $1 \leq k \leq d$ when the maximum or minimum point with respect to that dimension crosses the corresponding cell boundary in that dimension. These events are called *minor* events. Additionally, we have *major* events that occur when an event causes to change the movement of the bounding cube. At major events the whole data structure is updated as defined above.

Every event in a kinetic turnament tree is called a turnament event.

**Time to process events.** We first consider minor events. In every minor event, a point $p$ in some set $S_{i,j}$ moves from one cell $C_1$ of the grid into another cell $C_2$. Therefore, $p$ is deleted from $2d$ turnament trees corresponding to $C_1$ and is inserted into the $2d$ turnament trees corresponding to $C_2$. If the point moves into a cell that was previously empty, we must insert the cell index of $C_2$ into the search tree $T_{i,j}$ and initialize the $2d$ turnament trees. If $p$ was the only point in $C_1$ we have to delete the $2d$ turnament trees. Since (cf. Section 7.1) in $O(\log n)$ time one can insert a point in a turnament tree or search tree and since any insertion in a kinetic turnament tree creates $O(\log n)$ new events in expectation, we get:

**Lemma 7.3.1** *Any minor event can be processed in* $O(d \log^2 n)$ *time. It creates* $O(d \log n)$ *new events in kinetic turnament trees in expectation.*

Next we want to analyze the running time required to setup the whole data structure at major events.

**Lemma 7.3.2** *Any major event can be processed in expected time* $O(d \cdot K \cdot n \cdot \log n)$

**Proof :** The time to setup our data structure at a major event is dominated by the time to setup the kinetic turnament trees for the boundary events. Since each kinetic turnament tree consisting of $m$ points can be constructed in time $O(m \cdot \log m)$ we have to count the number of sample points in all kinetic turnament trees. Each sample point is inserted into $2d$ kinetic turnament trees. The expected number of points in $S_{i,j}$ is $Kn/2^{i+j}$. By linearity of expectation we get that the total number of points in all kinetic turnament trees is

$$\sum_{i,j} 2d \cdot \frac{Kn}{2^{i+j}} = O(dKn) \ .$$

□

**Remark 7.3.3** *Instead of setting up the data structure at major events we can precompute the major events in time* $O(d^2 \cdot n)$. *Then in time* $O(d^2 \cdot n)$ *we can precompute all times of major events and the corresponding scaled positions and velocities of the points. From that we can setup all data structures at the beginning instead of setting them up at major events. Using this technique we get a total expected setup time of* $O(d^3 \cdot K \cdot n \cdot \log n)$ *and can process major events in time* $O(1)$ *by just switching to the precomputed structure.*

**Analysis of the number of events.** In Section 7.2 we showed:

**Lemma 7.3.4** *There are at most* $O(d^2)$ *major events.* □

Since (after scaling the points and velocities at major events) the bounding cube $B = [0,1]^d$ is fixed between major events, it follows that for every grid $\mathcal{G}_i$ the boundaries of the grid cells are fixed as well between major events. Therefore we get:

**Lemma 7.3.5** *During the linear motion between major events every point crosses at most* $d \cdot (2^i - 1)$ *cells in grid* $\mathcal{G}_i$.

**Proof :** Let us consider an arbitrary point $p$. We regard the cell boundaries in each dimension separately. In grid $\mathcal{G}_i$ we have $2^i - 1$ internal boundaries. Since $p$ moves linearly in time, $p$ can cross each boundary at most once. Since this can happen in each of the dimensions, the lemma follows. □

**Corollary 7.3.6** *The expected number of minor events is* $O(d^3 K n \cdot Z)$.

**Proof :** The expected number of minor events involving points from $S_{i,j}$ is at most $\frac{Kn}{2^{i+j}} \cdot d \cdot 2^i = dKn/2^j$ between two major events. Summing up over all $i, j$ and multiplying it with the number $O(d^2)$ of major events, we get that there are at most $O(d^3 K n Z)$ events. □

**Corollary 7.3.7** *The expected number of turnament events is* $O(d^4 \cdot K \cdot n \cdot Z \cdot \log n)$.

**Proof :** Every minor event creates a number of $O(d \log n)$ new events in kinetic turnament trees. Linearity of expectation implies that the expected number of events in kinetic turnament trees is $O(d^4 \cdot K \cdot n \cdot Z \cdot \log n)$. □

**Flight plan updates.** In KDS it is typically assumed that at certain points of time the *"flight plan"* of an object can change. At every such flight plan update the data structure is notified that a point now moves in another direction (possibly at a different speed). Such a flight plan update compels us to update all events in the event queue that involve this particular point. In our case we distinguish between two types of points. First, there is a constant number of points on which flight plan updates change the data structure maintaining the bounding cube. If the movement of one of these points is changed we have to setup a whole new data structure in time $O(d^3 \cdot K \cdot n)$.

If the flight plan of any other point is updated the movement of the bounding cube and the scaling of the points stays the same. We simply have to update all events the point is involved in. Since it requires $O(\log^2 n)$ time to update a randomized kinetic turnament tree we have to compute the expected number of such kinetic turnament trees a point is involved in. Every point is stored in $2d$ kinetic turnament trees for each set $S_{i,j}$ it is contained in. We get

**Lemma 7.3.8** *Let $p \in P$ be an arbitrary point. Then $p$ is stored in $O(dK)$ kinetic turnament trees in expectation.*

**Proof :** The probability that $p$ is contained in set $S_{i,j}$ is at most $K/2^{i+j}$. Hence, we get that the expected number of sets $S_{i,j}$ that contain $p$ is

$$\sum_{i,j} \frac{K}{2^{i+j}} \;=\; \sum_{i} \frac{1}{2^i} \sum_{j} \frac{K}{2^j} \leq \sum_{i} \frac{2K}{2^i} \;=\; O(K) \;.$$

The lemma follows from the fact that every point is stored in $2d$ kinetic turnament trees for each set $S_{i,j}$ it is contained in. □

Assume we fix some point of time and specify for each point an arbitrary flight plan update. If we choose one of these updates uniformly at random then the expected time to perform the update is small, i.e., the average cost of a flight plan update is low.

**Lemma 7.3.9** *A flight plan update can be done in $O(d \cdot (d^3 + \log n) \cdot \log n \cdot K)$ average expected time.*

**Proof :** It requires $O(\log^2 n)$ time to do a flight plan update of a kinetic turnament tree. In expectation every point is stored in $O(dK)$ kinetic turnament trees. Hence the expected time required to update these turnament trees is $O(dK \log^2 n)$. Additionally, we have to deal with updates of the $O(d)$ points that change the bounding cube structure. We can process such an update in $O(d^3 \cdot K \cdot n \cdot \log n)$ time. Averaging over all points we get that the average expected update time for these kind of updates is $O(\frac{d}{n} \cdot d^3 K n \log n) = O(d^4 \cdot K \cdot \log n)$. □

**Extracting the coreset.** We describe how to extract a small coreset from our data structure. We try to find a value of $j$ such that the coreset constructed with the value $\delta = \delta(j)$ has size of at most $S = O(\log n/\epsilon^{d+1})$ (Corollary 5.6.15), and that the size of the coreset constructed with $\delta = \delta(j-1)$ is greater than $S$. By Corollary 5.6.15 and Lemma 5.6.18 this coreset is then with probability $1 - \psi/j_0$ an $\epsilon$-coreset for MaxCut.

Testing the coreset size for one value of $j \in \{0, \ldots, j_0\}$ and extracting one coreset of size $S$ can be done in time $O(\log^2 n/\epsilon^{d+1})$:

Since (according to the proof of Lemma 5.4.7) there are at most $O(\log n/\epsilon^{d+1})$ heavy cells in each grid $\mathcal{G}_i$, we have a bound of $O(\log^2 n/\epsilon^{d+1})$ on the number of heavy cells. The marking process yielding to the actual coreset can be seen as a quadtree traversal. Since each inner node of this tree corresponds to a heavy cell, the tree traversal can be done in time $O(\log^2 n/\epsilon^{d+1})$. When we see that we have too many heavy cells we stop the process.

We can search for the right value of $j$ using a binary search. This takes a total time of $O(\log^2 n \cdot \log\log n/\epsilon^{d+1})$. Since we construct less than $j_0$ coresets, the whole process works with probability $1-\psi$.

**Computing an approximate MaxCut from the coreset.** According to Section 5.5.3 we can compute a $(1\pm\epsilon)$-approximate MaxCut-solution from the coreset in $O\left(\log^2 n \cdot 2^{((1/\epsilon)^{O(1)})}\right)$ time.

We can apply this algorithm to the computed coreset and obtain our main result. In the theorem we assume that $d$ is a constant.

**Theorem 23** *There is a kinetic data structure that maintains a $(1+\epsilon)$-approximation for the Euclidean MaxCut problem, which is correct with probability $1-\psi$. The data structure answers queries of the form 'to which side of the partition belongs query point $p$?' in $O(\log^2 n \cdot \log\log n \cdot \epsilon^{-2(d+1)} \cdot 2^{1/\epsilon^{O(1)}})$ time. Under linear motion the data structure processes expected $\widetilde{O}(\frac{n\log(\psi^{-1})}{\epsilon^{d+3}})$ events, which require $O(\log^2 n)$ time. A flight plan update can be performed in $\widetilde{O}(\frac{\log^4 n\cdot\log(\psi^{-1})}{\epsilon^{d+3}})$ average expected time, where the average is taken over the worst case update times of the points at an arbitrary point of time. The data structure needs an expected setup time of $\widetilde{O}(\frac{n\cdot\log(\psi^{-1})}{\epsilon^{d+3}})$.*□

# 8 An Efficient $k$-Means Implementation using Coresets

In this chapter we develop an efficient k-means clustering algorithm called CoreMeans. The main new idea is that the algorithm uses the coreset construction of Section 5.3 to speed up the computation of the clustering. Our algorithm first computes a small coreset of the input points and then runs variant of KMHybrid [104, 83], which is a combination of Lloyd's algorithm and random swaps. Then the algorithm doubles the size of the coreset and runs for a few steps on this coreset. This process is done until the coreset coincides with the whole point set. The coreset computation is supported by a quadtree (or higher dimensional equivalent) based data structure. This data structure can also be used to speed up nearest neighbor queries.

We will compare our algorithm with algorithm KMHybrid [104, 83] in Section 8.3. On most of the input instances our algorithm significantly outperforms KMHybrid, especially for low dimensional instances. For high dimensional instances our algorithm finds good solutions faster but KMHybrid's solution after a few seconds is slightly better. If we want to compute a clustering for one value of $k$ the running time of both algorithms is often dominated by the setup time to compute auxiliary data structures. In this case CoreMeans benefits from its smaller setup time.

However, in many applications we do not know the right value of $k$ in advance. In such a case one has to compute clusterings for many different values of $k$. Then one can use a quality measure independent of $k$ to find out the best clustering. A prominent quality measure for such a scenario is the *average silhouette coefficient* [86]. Although there are no theoretical guarantees for the average silhouette coefficient, it is often used to evaluate the quality of different clusterings. Unfortunately, computing the *average silhouette coefficient* for one clustering takes time quadratic in the number of points, which is not feasible for point sets of medium and large size. However, we would like to compute the sihouette coefficient for many values of $k$.

In this situation we can see the real strength of coresets. Using coresets it is possible to find clusterings and compute their average silhouette coefficient for large point sets and many values of $k$. For example, we computed clusterings for $k = 1, \ldots, 100$ and approximated their average silhouette coefficient for a set of more than $4.9$ million points in 3D consisting of the RGB values of an image in a few seconds on one core of an Intel Pentium D dual core processor with 3 GHz core frequency. In higher dimensions we did the same computations for an (artificially created) point set of $300,000$ points in $20$ dimensions for all values of $k$ between $1$ and $100$ in less than $8$ minutes. Without coresets, the computation of the sihouette coefficient even for one value of $k$ takes several hours.

First, we develop some notation and introduce the basic k-means method.

## 8.1 Definitions and Notations

In this chapter we will deal with weighted and unweighted sets of points. We will always assume that the set $P$ represents the input instance, which is an unweighted set of $n$ points in the $\mathbb{R}^d$. A weighted point set will usually be denoted by $R$. The weights of the points in $R$ are given by $w(r)$ for every $r \in R$. We will only consider integer weights in this paper. For each point $p \in \mathbb{R}^d$ let $p^{(j)}$ denote it's $j$-th coordinate.

Recall the definition of the $k$-means clustering problem from Section 2.3.

Two easy characterizations of an optimal $k$-means solution are known. We state them in Lemma 8.1.1 and Lemma 8.1.2:

**Lemma 8.1.1** *Let $C = \{c_1, \ldots, c_k\}$ be a set of fixed cluster centers with $|C| = k$. Let $C_1, \ldots, C_k$ be a partition of $P$ which fulfills*

$$p \in C_i \;\Rightarrow\; \forall_{j \in \{1,\ldots,k\}} d(p, c_i) \leq d(p, c_j) \ .$$

*Then the partition $C_1, \ldots, C_k$ minimizes the $k$-means objective function for the fixed set of centers $C$.* □

We write $\mathit{Means}(P, C)$ to denote the cost of an optimal partition of $P$ with respect to the centers in $C$. In a similar way we write $\mathit{Means}(R, C)$ to denote the cost of an optimal partition of a weighted set $R$ with respect to $C$.

The other way around we can easily find the best centers for a given partition. The result is also well known (and proven in many publications like [94]).

**Lemma 8.1.2** *Let $C_1, \ldots, C_k$ be a fixed partition of $P$. For all $i \in \{1, \ldots, k\}$ let $c_i := \mu(C_i)$ be the center of gravity of $C_i$ as defined in Section 2.1. Then the centers $c_1, \ldots, c_k$ minimize the $k$-means objective function for the fixed partition $C_1, \ldots, C_k$ of $P$.*

**Proof :** We can do the minimization within each cluster seperately. Therefore we have to show for a set of points $Q$, that the function $f : \mathbb{R}^d \to \mathbb{R}$ with $f(r) = \sum_{q \in Q} d(q, r)^2$ is minimized at $r = \mu(Q)$.

We have

$$f(r) = \sum_{q \in Q} d(q, r)^2 = \sum_{q \in Q} \sum_{i=1}^{d} (q^{(i)} - r^{(i)})^2 = \sum_{i=1}^{d} \sum_{q \in Q} (q^{(i)} - r^{(i)})^2 \ .$$

Therefore we can minimize each dimension seperately: For $i \in \{1, \ldots, d\}$ let

$$f_i(x) = \sum_{q \in Q} (q^{(i)} - x)^2 \ .$$

When we chose $r^{(i)}$ such that $f_i(r^{(i)}) = \sum_{q \in Q} (q^{(i)} - r^{(i)})^2$ is minimized, then $r$ minimizes $f$.

Let $i \in \{1, \ldots, d\}$ be fixed. Obviously $f_i : \mathbb{R} \to \mathbb{R}$ is continuously differentiable. Furthermore $\lim_{x\to\infty} f_i(x) = \infty$ and $\lim_{x\to-\infty} f_i(x) = \infty$. Therefore $f_i$ must have a global minimum, and at that global minimum we must have $f'(x) := \frac{d}{dx}f(x) = 0$.

We observe:

$$f'(x) = \sum_{q\in Q} 2 \cdot (x - q^{(i)}) \ .$$

Therefore we have

$$\begin{aligned} f'(x) = 0 \ &\Leftrightarrow \ \sum_{q\in Q} 2 \cdot (x - q^{(i)}) = 0 \Leftrightarrow \left(\sum_{q\in Q} x\right) - \left(\sum_{q\in Q} q^{(i)}\right) \\ &\Leftrightarrow \ |Q| \cdot x = \sum_{q\in Q} q^{(i)} \Leftrightarrow x = \frac{1}{|Q|}\sum_{q\in Q} q^{(i)} \Leftrightarrow x = \mu(Q)^{(i)} \ . \end{aligned}$$

Therefore $\mu(Q)$ is the only point in $\mathbb{R}^d$ minimizing f. □

### 8.1.1 The Basic $k$-Means Method

Based on the observations of Lemma 8.1.1 and Lemma 8.1.2 that it is easy to compute an optimal partition for a fixed set of centers and an optimal set of centers for a fixed partition, a simple and elegant clustering heuristic has been developed [41, 95, 97]. Nowadays, one often refers to this heuristic as *the* $k$*-means algorithm* or *Lloyd's algorithm*. This algorithm runs in iterations. At the beginning of an iteration the algorithm has a set of $k$ centers $\{c_1, \ldots, c_k\}$. Every iteration consists of two steps:

1. For every $p \in P$ compute its nearest center in $\{c_1, \ldots, c_k\}$. Partition $P$ into $k$ sets $C_1, \ldots, C_k$ such that $C_i$ contains all points whose nearest center is $c_i$ (and break ties arbitrarily).

2. For every cluster $C_i$ compute its center of gravity $\mu(C_i)$, i.e. the optimal center of that cluster. Then set $c_i := \mu(C_i)$ for every $1 \leq i \leq k$.

Each iteration runs in $O(ndk)$ time. Typically, the algorithm runs for a fixed number of iterations (standard values are in the range from 50 to 500). It is well known that the algorithm only converges to a local optimum and that the quality of this solution depends strongly on the initial set of centers. Therefore, the algorithm is usually repeated several times with different sets of initial centers and the best discovered partition is returned.

### 8.1.2 Algorithm KMHybrid

In the experiments we compare our algorithm to an algorithm called KMHybrid [104, 83]. KMHybrid combines Lloyd's algorithm with swapping of centers (moving centers away to the position of a random point to break local minima) and a variant of simulated annealing. The algorithm does one swap followed by a sequence of Lloyd's steps and an acceptance test. If the

current solution passes the test, the algorithm continues with the current solution. Otherwise, it restores the old one. The acceptance test is based on a variant of simulated annealing. Additionally, the algorithm uses a kD-tree to speed up nearest neighbor search in the Lloyd's steps. For more details see [104, 83].

### 8.1.3 The Silhouette Coefficient

In many applications the right number of clusters is not known in advance. Since the k-means objective function drops monotonically as $k$ increases, one needs a different measure for the quality of a clustering that is independent of $k$. Such a measure is provided by the average *silhouette coefficient* [86] of the clustering. The silhouette coefficient of a point $p_i$ is computed as follows. First compute the average distance of $p_i$ to the points in the same cluster as $p_i$. Then for each cluster $C$ that does not contain $p_i$ compute the average distance from $p_i$ to all points in $C$. Let $b_i$ denote the minimum average distance to these clusters. Then the silhouette coefficient of $p_i$ is defined as $(b_i - a_i)/\max(a_i, b_i)$.

The value of the silhouette coefficient of a point varies between $-1$ and $1$. A value near $-1$ indicates that the point is clustered badly. A value near $1$ indicates that the point is well-clustered. To evaluate the quality of a clustering one can compute the average silhouette coefficient of all points.

## 8.2 The Algorithm

We first provide a high level description of our algorithm and then we give some more details on the implementation.

Our algorithm uses the observation that the first iterations of k-means algorithms or swapping heuristics do not need the accuracy of the whole point set. Our algorithm uses the coreset construction technique of Section 5.3 to reduce the complexity of the point set. The hope is that the first iterations of iterative algorithms can be done much more efficiently on small coresets, still significantly improving the objective function.

The algorithm starts to compute a coreset of size roughly $2k$ and chooses $k$ points from this coreset as a starting solution. Then it repeats $\max\{40/\sqrt{k}, 2\}$ times[1] the following two steps, which form the main loop of the algorithm.

- First it runs Lloyd's algorithm for $d$ steps. After this, the current solution is compared to the previously best and the algorithm continues with the better of these solutions.
- In the second step, the algorithm chooses a number $k_0$ between $0$ and $k$ uniformly at random. Then it picks centers from the current set of centers according to the following probability distribution until $k_0$ different centers are chosen. The probability that the center

[1] The value comes from the idea that the time to extract the coreset should be comparable to the time to run iterations on the coreset. Without a dependency on $k$ here, the coreset extraction time dominates the algorithm runtime for small values of $k$ at the beginning of the algorithm. The special value $40/\sqrt{k}$ has been empirically adjusted.

```
CoreMeans(P, k)
  m ← 2k
  while m ≤ |P| do
    Compute coreset of size m.
    if m = 2k then
      C ← k random points from coreset
      K ← C
    repeat max{40/√k, 2} times      (* Main loop *)
      Do d iterations of Lloyd's algorithm starting with C
      Let C denote the current solution
      if Means(P, K) < Means(P, C) then
        C ← K
      K ← C
      Choose k_0 randomly from {0, ..., k}
      Swap k_0 centers from C with points chosen uniformly from the coreset
    m ← 2 · m
```

Figure 8.1: The CoreMeans algorithm

of cluster $C_j$ is chosen is $\frac{1}{|C_j| \cdot \sum_{i=1}^{k} \frac{1}{|C_i|}}$, where $C_1, \ldots, C_k$ denotes the current clustering. Therefore centers of smaller clusters are picked with higher probability. Finally, these $k_0$ random centers are replaced by points chosen uniformly at random from the coreset.

After the main loop is finished the algorithm doubles the size of the coreset and continues with the main loop. This is done until the coreset is the whole point set. The algorithm is given in Figure 8.1.

To support efficient computation of coresets and to speed up nearest neighbor queries in Lloyd's algorithm we use a quadtree or its higher dimensional equivalent. Our approach is the analog to the kD-tree algorithm from [83]. The root of the quadtree corresponds to a bounding box of the point set. With each occupied cell $B$ associated with a node of the quadtree we store:

- The number $n_B = |P \cap B|$ of points contained in the cell,
- A vector $b_B = (b_B^{(1)}, \ldots, b_B^{(d)})$ with $b_B^{(i)} := \sum_{p \in P \cap B} p^{(i)}$,
- The sum of the squared $\ell_2$-norm of the point vectors $e_B := \sum_{p \in P \cap B} \|p\|_2^2 = \sum_{p \in P \cap B} \sum_{i=1}^{d} (p^{(i)})^2$.

This information can be used to quickly compute the *exact* cost of the partition of $P$ that corresponds to a given partition of the coreset. The same precomputations have also been used in [83, 124]. Since all points from one cell are assigned to the same coreset point and hence to the

same cluster, we can compute the cost of a cluster C in the following way. Let $B_1, \ldots, B_l$ be the disjoint cells assigned to C. We first compute the center of gravity $c := \mu(C)$ using the formula

$$\mu(C)^{(i)} = \frac{1}{|C|} \cdot \sum_{p \in C} p^{(i)} = \sum_{j=1}^{l} \sum_{p \in P \cap B_j} p^{(i)} = \sum_{j=1}^{l} b_{B_j}^{(i)} \ .$$

Then we have to compute

$$\begin{aligned} \sum_{p \in C} d(p,c)^2 &= \sum_{p \in C} \sum_{i=1}^{d} (p^{(i)} - c(i))^2 = \sum_{p \in C} \sum_{i=1}^{d} \left[ (p^{(i)})^2 - 2p^{(i)}c^{(i)} + (c^{(i)})^2 \right] \\ &= \sum_{j=1}^{l} e_{B_j} - 2 \cdot \sum_{i=1}^{d} c^{(i)} \cdot \sum_{j=1}^{l} b_{B_j}^{(i)} + \sum_{j=1}^{l} n_{B_j} \cdot \left( \sum_{i=1}^{d} (c^{(i)})^2 \right) \ , \end{aligned}$$

which can be done efficiently by using the information stored with each cell. Notice that the runtime of this computation only depends on the number of cells, not on the number of points inside the cells.

In our implementation we fixed the depth of the quadtree to 11[2]. To build the tree we proceeded bottom-up. We identify the non-empty cells in the grid corresponding to the 11-th level of the tree. The non-empty cells are stored in a hash table with cell coordinates as keys. After we have computed the non-empty cells in the finest grid we iterate over these cells and compute all non-empty cells in the next coarser grid together with the corresponding point statistics.

To compute a coreset of around m points we first have to identify a good guess for $\delta$. Larger values of $\delta$ lead to a smaller coreset but also to a worse approximation (see Section 5.3). We find a good value by setting $\delta$ to a large value and dividing it iteratively by 1.3. After each iteration we compute the heavy cells from the cell statistics, and from that the size of the coreset (without computing the actual coreset points). For high values of $\delta$ this is done very fast since the coreset size can be computed using a few large cells. Alltogether the time to compute the coreset *size* is negligible compared to the coreset computation time.

The coreset for a given value of $\delta$ is computed using a recursive depth first search function on the quadtree cells. For the root cell we call a function COMPUTECORESETPOINTS (see Figure 8.2). This function has a cell as input parameter as well as statistics about points to be moved into that cell. COMPUTECORESETPOINTS adds the input points to the cell statistics. Then for each subcell it checks heaviness. If there is at least one heavy subcell, it calls COMPUTECORESETPOINTS for all heavy subcells. The points given as function parameter and the points in all light subcells are then moved into a heavy subcell by adding up their statistics and giving these statistics to one call of COMPUTECORESETPOINTS as function parameters. If a cell has no

[2]During the computation of big coresets in later iterations it can happen that the coreset extraction needs point information of deeper levels. However, since we used images as test instances with 8 bits per color channel, a depth of 11 was enough. We believe that this also suffices for most other real world scenarios.

**Global variables** used in functions COMPUTECORESETPOINTS, COMPUTEASSIGNMENTFORCELL, AND COMPUTEASSIGNMENT

$L$ : the current set of centers.
For each center $c \in L$:
$n_c \in \mathbb{N}$ : the number of points assigned to the center.
$b_c =\in \mathbb{R}^d$ : the sum of coordinates of points assigned to the center.
$e_c =\in \mathbb{R}$ : the sum of squared $\ell_2$-norms of points assigned to the center.
$R$ : the coreset computed by COMPUTECORESETPOINTS.
For each cell $C$ with points in it:
$n_C \in \mathbb{N}, b_C \in \mathbb{R}^d, e_C \in \mathbb{R}$: The statistics for the points $P \cap C$ in cell $C$.
$\widetilde{n}_C \in \mathbb{N}, \widetilde{b}_C \in \mathbb{R}^d, \widetilde{e}_C \in \mathbb{R}$: The statistics for the points $P_C$ in cell $C$ taking into account the movement of points into cells during coreset construction.

COMPUTECORESETPOINTS returns a set of coreset points, each being a triple from $\mathbb{N} \times \mathbb{R}^d \times \mathbb{R}$.
COMPUTECORESETPOINTS(Cell $C, n \in \mathbb{N}, b \in \mathbb{R}^d, e \in \mathbb{R}$)
$\widetilde{n}_C \leftarrow n_C + n$.
$\widetilde{b}_C \leftarrow b_C + b$.
$\widetilde{e}_C \leftarrow e_C + e$.
Let $H_1, \ldots, H_h$ be the heavy subcells of $C$.
Let $L_1, \ldots, L_l$ be the light subcells of $C$.
**for** j=1,...l **do:**
$\widetilde{n}_{L_j} \leftarrow 0$. // Points are moved into a heavy cell.
$\widetilde{b}_{L_j} \leftarrow (0, \ldots, 0) \quad \in \mathbb{R}^d$.
$\widetilde{e}_{L_j} \leftarrow 0$.
$N \leftarrow n + \sum_{j=1}^{l} n_{L_j}$. // Number of points to be assigned.
$B \leftarrow b + \sum_{j=1}^{l} b_{L_j}$.
$E \leftarrow e + \sum_{j=1}^{l} e_{L_j}$.
**if** $C$ has at least one heavy subcell **do:**
Coreset $\leftarrow$ COMPUTECORESETPOINTS$(H_1, N, B, E)$.
**for** $j = 2, \ldots, h$ **do:**
Coreset $\leftarrow$ Coreset $\cup$ COMPUTECORESETPOINTS$(H_j, 0, \overrightarrow{0}, 0)$.
**else**
Coreset$\leftarrow \{(N, B, E)\}$. // Create new coreset point.
RETURN Coreset.

Figure 8.2: The COMPUTECORESETPOINTS function

heavy subcell, a coreset point is introduced from the statistics about cell points and the statistics given as function input.

To speed up of the later k-means algorithm we store the following statistics: For each cell occupied by coreset points we store a pointer to a corresponding coreset point (if there is one) and pointers to all subcells containing coreset points. For a cell $B$ let $P_B$ be the points from $P$ which are in cell $B$ after moving points during the coreset construction. During the construction of coreset $R$ we additionally compute for each cell $B$ the corresponding coreset statistics (i.e. the cell statistics taking into account that points are moved into cells during the coreset construction):

- The number $\widetilde{n}_B = |P_B|$ of points contained in the cell (after moving points).
- A vector $\widetilde{b}_B = (\widetilde{b}_B^{(1)}, \ldots, \widetilde{b}_B^{(d)})$ with $\widetilde{b}_B^{(i)} := \sum_{p \in P_B} p^{(i)}$,
- The sum of the squared $\ell_2$-norm of the point vectors $\widetilde{e}_B := \sum_{p \in P_B} \|p\|_2^2 = \sum_{p \in P_B} \sum_{i=1}^{d} (p^{(i)})^2$.

See Figure 8.2 for details. The function COMPUTECORESETPOINTS returns a set of triples from $\mathbb{N} \times \mathbb{R}^d \times \mathbb{R}$, where triple $(n', b', e')$ stands for a coreset point representing $n'$ points from the input instance with coordinate sum $b'$ and sum of squared $\ell_2$-norm $e'$.

One iteration of the k-means algorithm is done as follows: Instead of searching the nearest center for each coreset point separately we use an approach analogous to the kd-tree approach as in [83]. We start with a list $\mathcal{L}$ of all centers as possible candidates for nearest centers and do a depth first walk on those quadtree cells which contain a coreset point. For each cell $C$ in the quadtree we check if we can rule out that some centers $q$ in $\mathcal{L}$ are the nearest center to any point in $C$. This is done by first computing the point $l$ from $\mathcal{L}$ that is nearest to the center of $C$. Then we check for each point $q \in \mathcal{L}$, whether $C$ lies completely on the same side as $l$ of the bisector between $l$ and $q$. All centers which cannot be nearest centers for coreset points in $C$ are evicted from $\mathcal{L}$ and the algorithm proceeds to the children of cell $C$.

If $|\mathcal{L}| = 1$ we know the nearest center for all coreset points within the cell. Since we hold statistics for all coreset points within each cell we can then assign all coreset points in one step to the center $l \in \mathcal{L}$ and stop.

If $|R \cap C| = 1$, we compute the distances of the coreset point to all $l \in \mathcal{L}$ directly, assign the coreset point and stop the depth first walk.

See Figure 8.3 for details.

**Computation of silhouette coefficients.** The computation of silhouette coefficients for each point $p_i$ is speeded up in the following way: We first compute the average distance $a_i$ to all points in the same cluster. To compute $b_i$, the minimum over average distances $b_{i,j}$ to points in other clusters $C_j$, we identify the second nearest cluster center $c_l$ and compute the average distance $b_{i,l}$ to all points in $C_l$. In most cases $b_{i,l}$ is the minimum of all $b_{i,j}$ for other clusters. To

```
ComputeAssignmentForCell(Cell C, List of possible centers $\mathcal{L} \subset L$)
  if $|R \cap C| = 1$ do: // Only one coreset point in C.
    Let $g \in R \cap C$ be the coreset point in cell C.
    Compute center $l \in \mathcal{L}$ nearest to g.
    $n_l \leftarrow n_l + \widetilde{n}_C$.
    $b_l \leftarrow b_l + \widetilde{b}_C$.
    $e_l \leftarrow e_l + \widetilde{e}_C$.
  else: // more than one coreset point in C.
    Let a be the center of cell C.
    Compute the center $l \in \mathcal{L}$ having smallest distance to a.
    for each $q \in \mathcal{L}$ :
      if each point of cell B has smaller distance to l than to q do:
        $\mathcal{L} \rightarrow \mathcal{L} \setminus \{q\}$.
    if $|\mathcal{L}| = 1$ do: // Then $\mathcal{L} = \{l\}$.
      $n_l \leftarrow n_l + \widetilde{n}_C$.
      $b_l \leftarrow b_l + \widetilde{b}_C$.
      $e_l \leftarrow e_l + \widetilde{e}_C$.
    else: // $|\mathcal{L}| \geq 2$.
      for each subcell $\widetilde{C}$ of C having a coreset point do:
        ComputeAssignmentForCell($\widetilde{C}, \mathcal{L}$).

ComputeAssignment()
  for each $l \in L$ do:
    $n_l \leftarrow 0$.
    $b_l \leftarrow (0, \ldots, 0) \in \mathbb{R}^d$.
    $e_l \leftarrow 0$.
  ComputeAssignmentForCell (Largest cell C, L).
```

Figure 8.3: The ComputeAssignment function

```
Silhouette(K)
  $m \leftarrow 5$.
  while $m \leq K$ do:
    Compute coreset of size m.
    for each $k \in \{1, \ldots, 100\}$ do
      Use main loop of CoreMeans to compute clustering.
      Compute average silhouette coefficients for the current coreset and centers.
    $m \leftarrow 2 \cdot m$.
```

Figure 8.4: The Silhouette function

| Instance | Size | Setup Times KMHybrid | CoreMeans |
|---|---|---|---|
| Tower | 4,915,200 | 28.59 | 4.77 |
| Bridge | 3,145,728 | 18.13 | 2.95 |
| PaSCO | 3,145,728 | 19.41 | 4.29 |
| Frymire | 1,234,390 | 4.71 | 0.65 |
| Clegg | 716320 | 2.76 | 1.05 |
| Monarch | 393,216 | 1.43 | 0.63 |
| Artificial5D | 300,000 | 2.27 | 1.49 |
| Artificial10D | 300,000 | 3.71 | 2.17 |
| Artificial15D | 300,000 | 4.71 | 2.70 |
| Artificial20D | 300,000 | 6.09 | 3.87 |

Table 8.1: Data sets and setup time.

get a certificate for this, we use the lower bound $d(p_i, c_j) \leq b_{i,j}$. We check for all other clusters if $d(p_i, c_j) \geq b_{i,l}$. If this inequality holds then $b_{i,l} \leq b_{i,j}$ and $b_{i,j}$ cannot be the minimal one. In that case we save the computations of all distances to points in cluster $C_j$.

## 8.3 Experiments

We implemented our algorithm using C++. The code was compiled using gcc version 3.4.4 using optimization level 2 (-O2). We compare our algorithm the implementation of KMHybrid from [83]. KMHybrid was compiled using the same compiler and also with optimization level 2. We ran it using the standard settings given by the developers.

We ran our experiments on an Intel Pentium D dual core processor with two cores. Both algorithms used only one core with core frequency 3 GHz. The computer has 2 GByte RAM.

### 8.3.1 Data Sets

We performed our experiments on two different types of instances. The first type of instance consists of images and we want to cluster the RGB values of the pixels. Thus the input points lie in 3D and the $i$-th input point corresponds to the RGB-values of the $i$-th pixel of the image. Such a clustering has applications in lossy data compression, since one can reduce the palette of colors used in the picture to the colors corresponding to the cluster centers.

Our test images consist of three large images (Tower, Bridge, and PaSCo) and three medium size images (Monarch, Frymire, and Clegg). The latter images are commonly used to evaluate the performance of image compression algorithms. The exact sizes of the test images can be found in Table 8.3.1. The images are available at homepages.uni-paderborn.de/frahling/coremeans.html

#### Artificially Created Instances.

The second type of instance is artificially created. Instance ArtificialxD consists of $300,000$ points in $x$ dimensions. The instance is generated by taking a random point from one of 20 Gaussian distributed clusters, whose centers are picked uniformly at random from the unit cube. The standard deviation of the Gaussian distribution is $0.02 \cdot \sqrt{d}$, i.e. it is the product of the one dimensional Gaussian distribution with standard deviation $0.02$. An example of a sample of points from instance Artificial2D is given in Figure 8.15.

### 8.3.2 Comparison of CoreMeans and KMHybrid

To evaluate the performance of CoreMeans we compare our algorithm to KMHybrid. We first compare the setup times for both algorithms, i.e. the time to construct the auxiliary data structures. If one wants to compute a clustering for fixed value of $k$ then the setup times often dominate the running time of the algorithm. If a good value of $k$ is not known, then one often wants to compute a clustering for multiple values of $k$. In this case, it is more interesting to compare the running time of both algorithms without setup time (however, the time to extract the coresets from the data structure is contained in the given running times). This is done in Sections 8.3.2 to 8.3.2. In Section 8.3.2 we compare both algorithm for different input sizes. In Section 8.3.2 we focus on the performance with increasing dimension, and in Section 8.3.2 we investigate into the dependence on the number of clusters.

#### Setup time

The times to compute the auxiliary data structure are given in Table 8.3.1. The time to build these structures does not depend on $k$. The setup time for KMHybrid is between $1.5$ to $7$ times higher than that of CoreMeans. There is a tendency that the gap becomes larger for larger instances. However, there seems to be also an dependence on the distribution of points as the largest factor was achieved for the medium size instance Frymire.

If one computes a clustering for one value of $k$ then the setup time is typically larger than the computation time. Even for the larger instances both algorithms obtain a good clustering in a few seconds (see also Section 8.3.2).

### Dependence on Input Size

To evaluate the dependence on the input size we run both algorithms on instance Monarch, Clegg, Frymire, PaSCo, Bridge, and Tower. We used paramter $k = 50$. In general, CoreMeans performs better for smaller $k$ (see Section 8.3.2) and tends to perform similar to KMHybrid as $k$ increases. The results are shown in Figures 8.5 to 8.10. The plots give the average performance of 10 runs. The vertical bars indicate the best and worst solution found within these runs. The relative performance of CoreMeans increases slightly with the size of $n$. We would like to emphasize that the difference between the best and worst solution found during the 10 runs is much smaller for CoreMeans. Therefore, to guarantee a good solution we have to run KMHybrid more often than CoreMeans. Another interesting observation is that CoreMeans achieves slightly better approximations for larger instances.

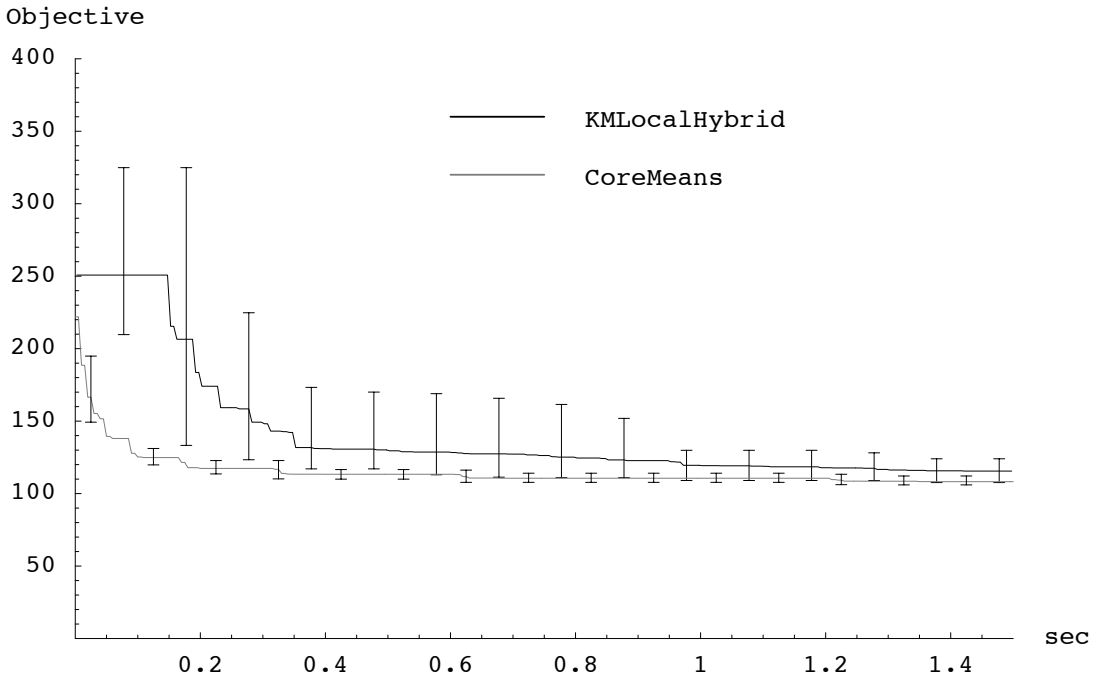


Figure 8.5: Performance on data set Monarch for $k = 50$ excluding setup time.

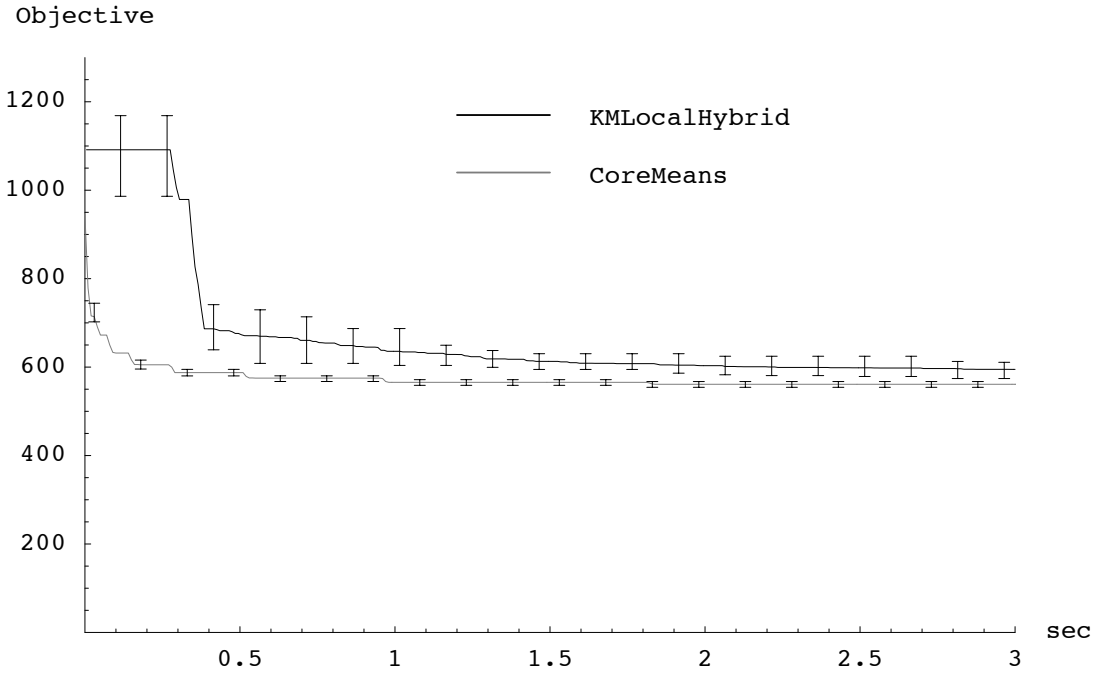


Figure 8.6: Performance for Clegg for $k = 50$ excluding setup time.

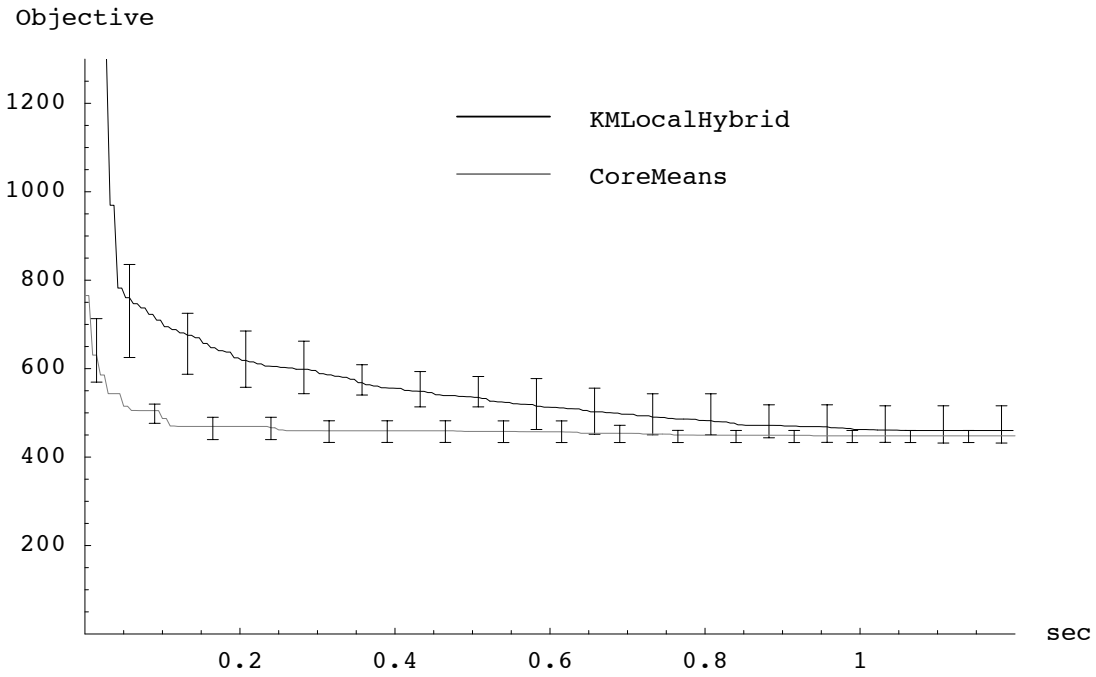


Figure 8.7: Performance for Frymire for $k = 50$ excluding setup time.

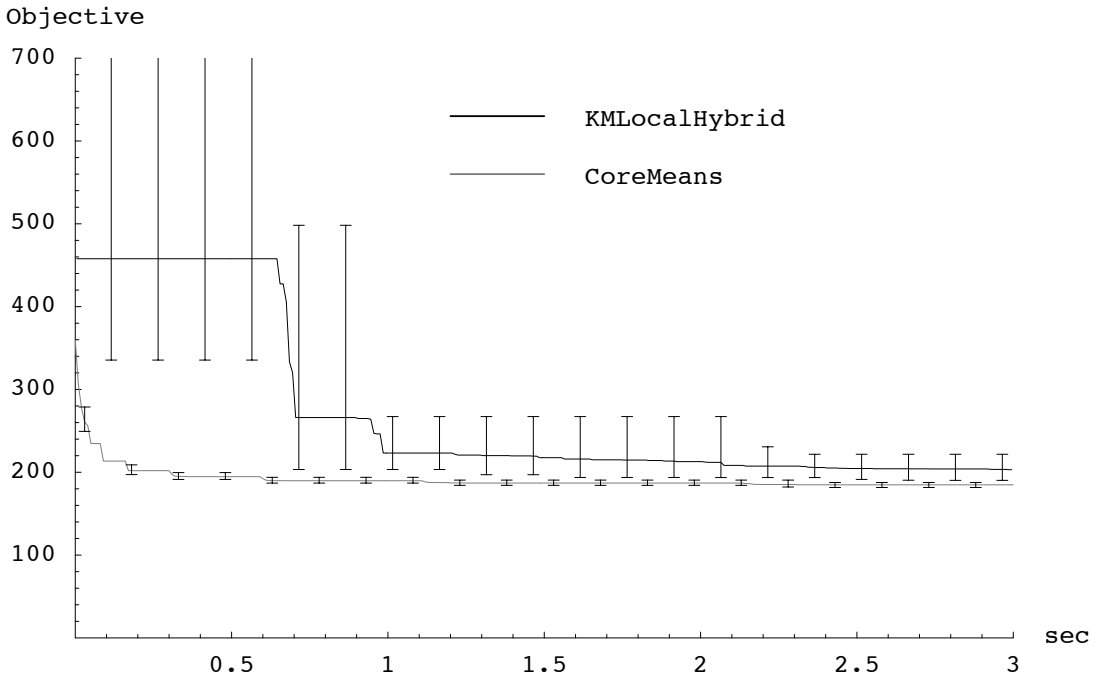


Figure 8.8: Performance for PaSCo for $k = 50$ excluding setup time.

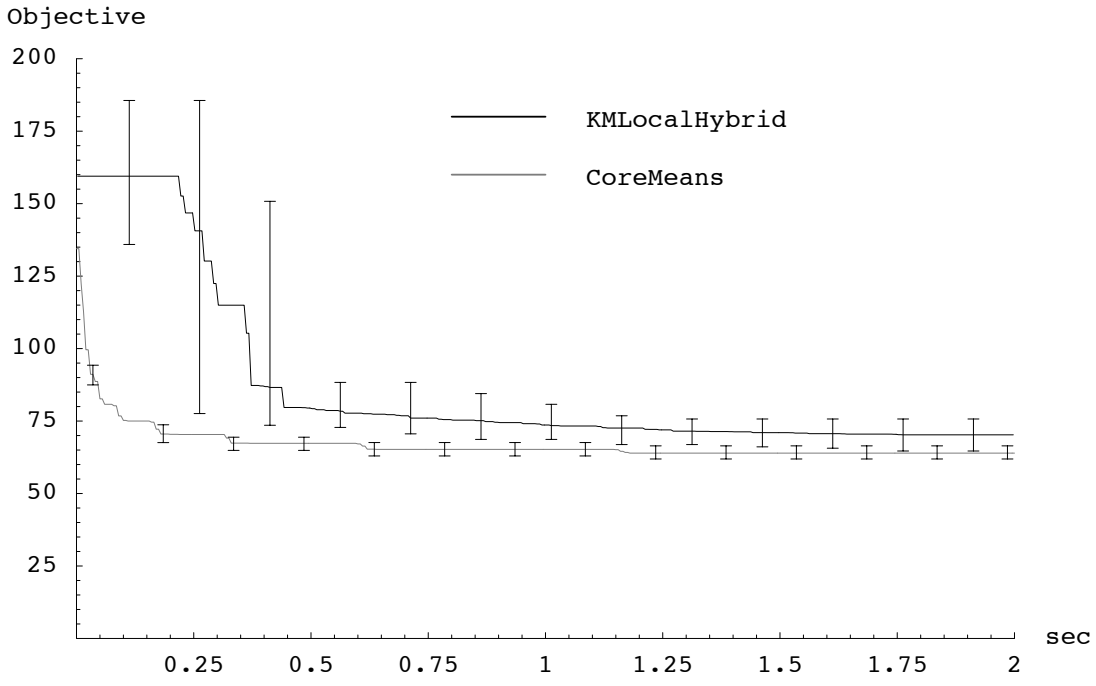


Figure 8.9: Performance for Bridge for $k = 50$ excluding setup time.

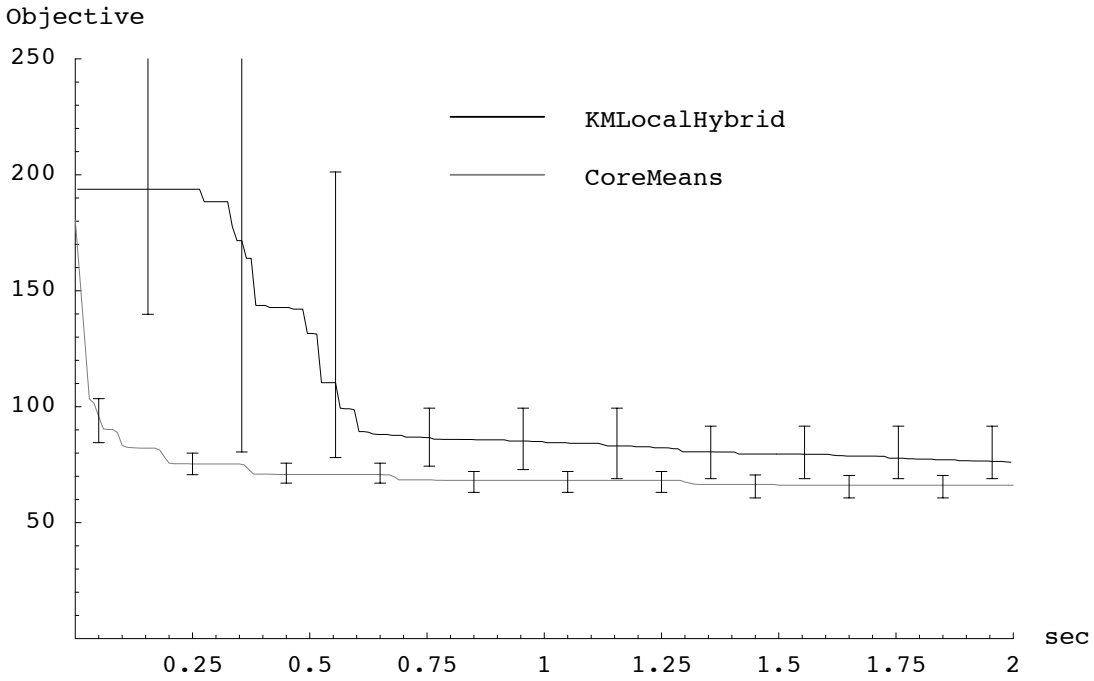


Figure 8.10: Performance for Tower for $k = 50$ excluding setup time.

### Dependence on the Dimension

Next we are interested in the dependence on the dimension. To evaluate this dependence, we compare the average performance of 10 runs of KMHybrid and CoreMeans for $k = 20$ on the instances ArtificialxD for $x = 5, 10, 15$, and 20. The graphs are shown in Figure 8.11 and 8.12. CoreMeans performs better on all instances. The most significant difference in performance can be found in the 5D instance, where CoreMeans performs a factor $10 - 30$ better. The higher the dimension the smaller is the advantage of CoreMeans. In these experiments the deviation of KMHybrid was much bigger than that of CoreMeans. Although CoreMeans shows the much better average performance, the best solution found by KMHybrid was better than the best solution found by CoreMeans. Overall, the performance of the algorithm for medium dimensions was much better than theory predicts with an exponential dependence on the dimension.

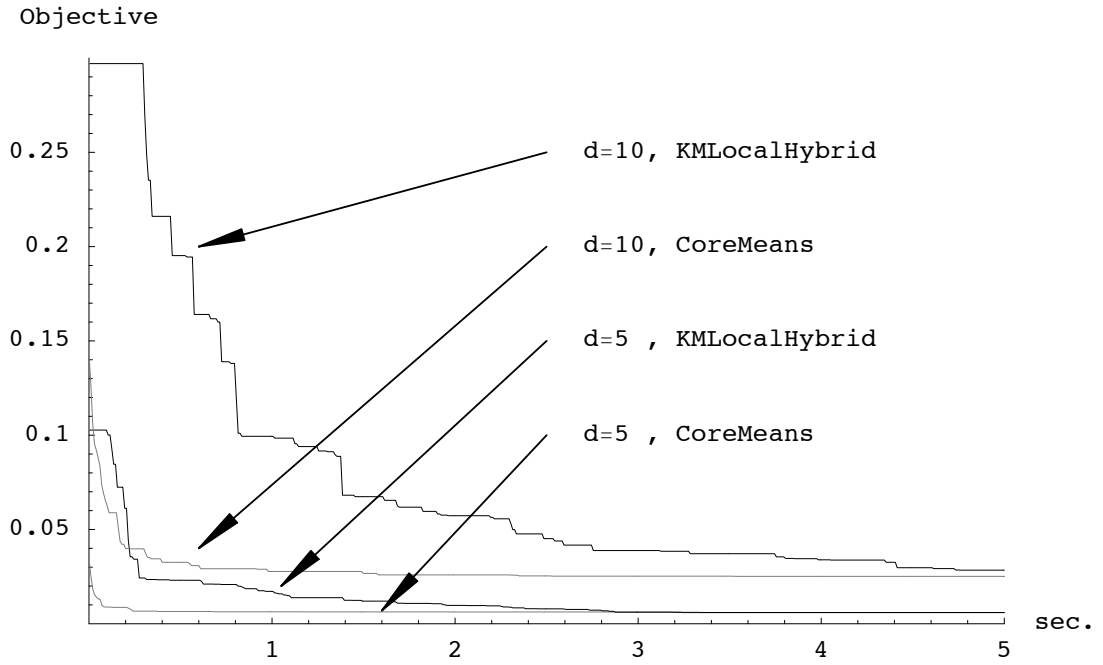


Figure 8.11: Performance for ArtificialxD with $x = 5$ and $x = 10$ excluding setup time.

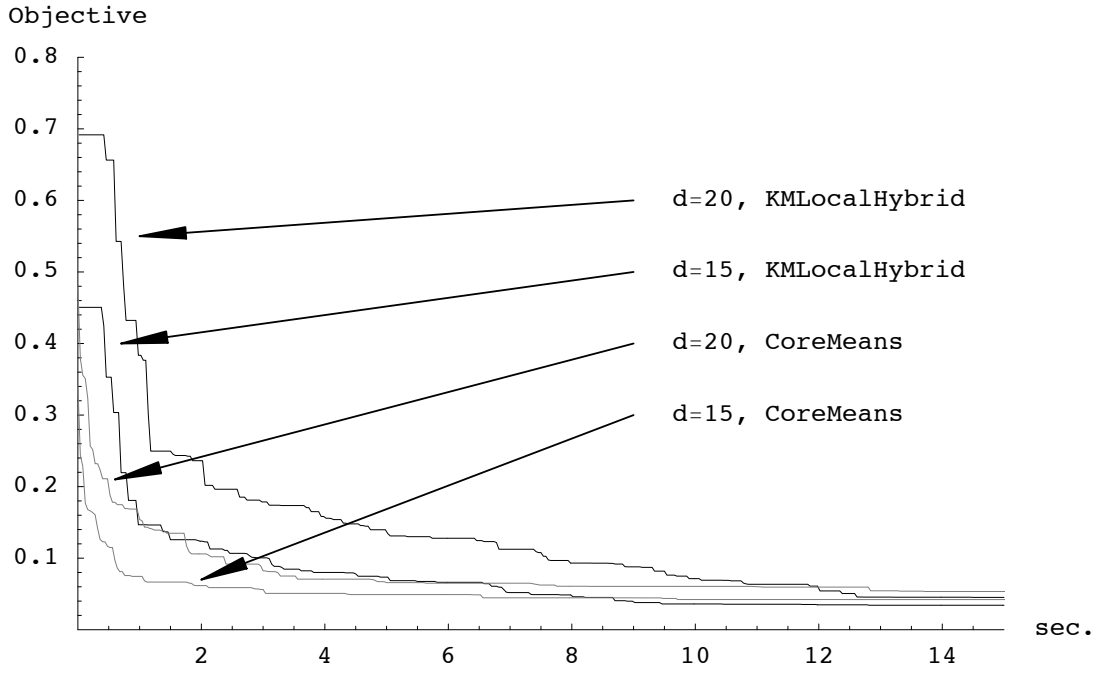


Figure 8.12: Performance for ArtificialxD with $x = 15$ and $x = 20$ excluding setup time.

### Dependence on the Number of Clusters

To investigate in the dependence on the number of cluster centers, we ran a number of experiments on different inputs. Due to space limitations we only present results for $k = 10, 50, 100$, and 200 for instance Bridge. These results are typical for the performance we encountered. As before, the Figures 8.13 and 8.14 show the average performance of 10 runs excluding the setup times. Typically, our algorithm performs significantly better for small values of $k$. For example, for $k = 10$ CoreMeans often performs a factor $10 - 100$ better. Additionally, the quality of the solutions computed by KMHybrid varies significantly. CoreMeans is less sensitive to the random choices of the algorithm. As a consequence one must perform more runs of KMHybrid to obtain a good solution with high probabilty. As $k$ grows larger the performance gap between the two algorithms decreases. The reason for this is that the quality of the coreset decreases as $k$ grows.

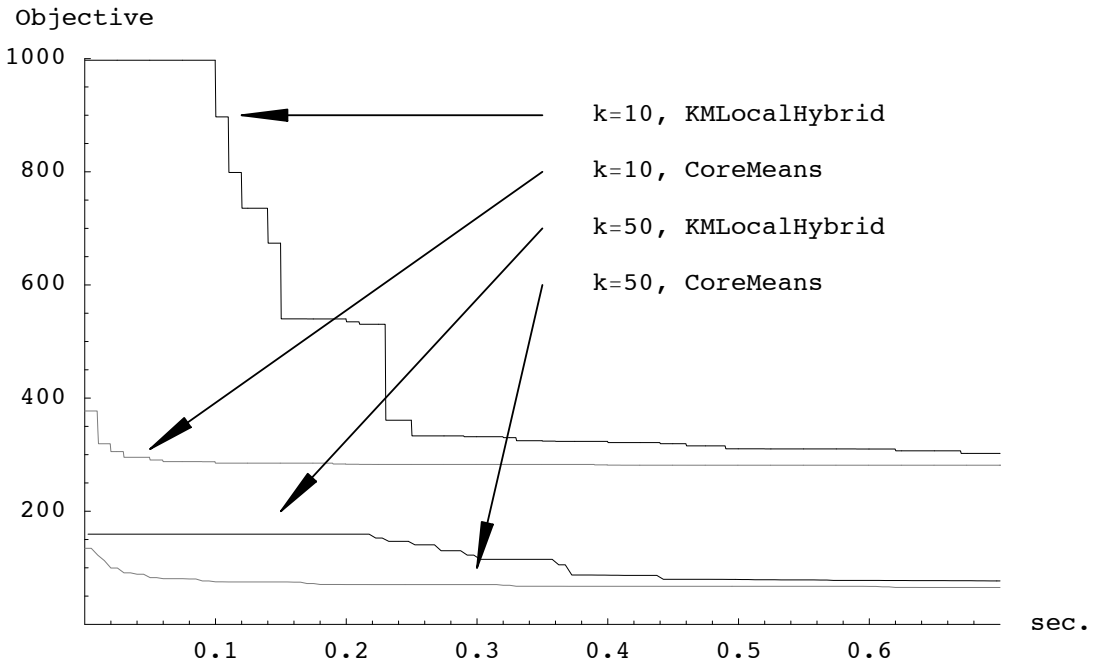


Figure 8.13: Performance for Bridge with $k = 10$ and $k = 50$ excluding setup time.

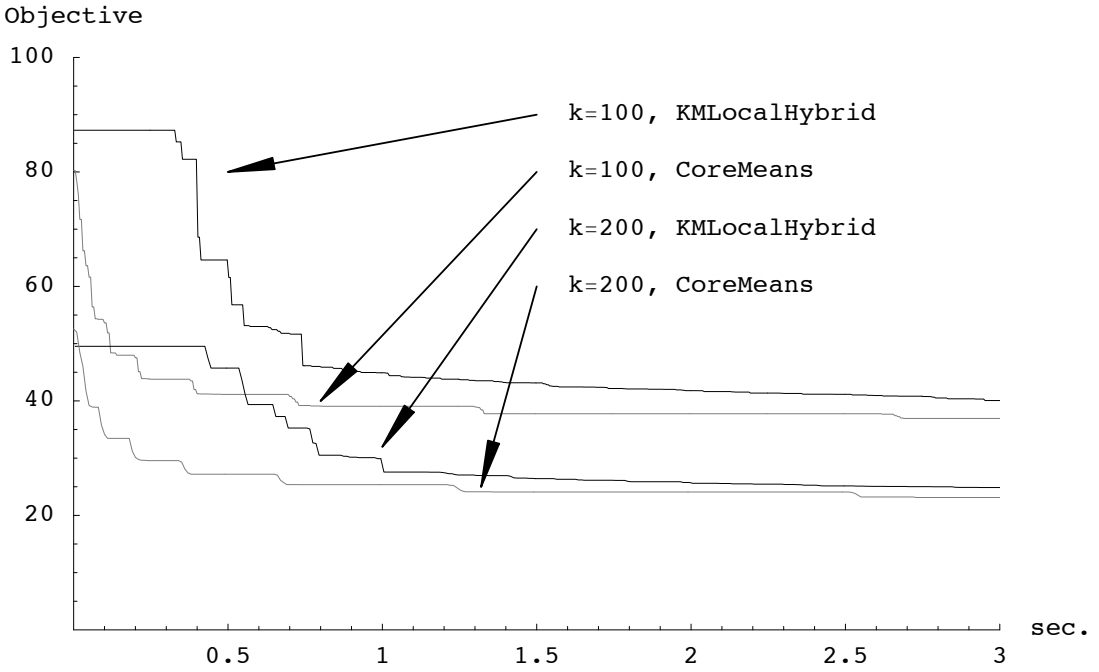


Figure 8.14: Performance for Bridge with $k = 100$ and $k = 200$ excluding setup time.

### 8.3.3 Computing the Silhouette Coefficient

We computed the approximate average silhouette coefficient for $1 \leq k \leq 100$ for instances Tower, Clegg, Monarch, and ArtificialxD with $x = 2, 10, 20$ using coresets of different sizes. Table 8.2 summarizes the running times of our tests. The second column gives the overall running time for the computation and the third column states the time spend to compute the silhouette coefficients. Since the time to compute the silhouette coefficient is quadratic in the coreset size, the fraction of time spent for this computation increases significantly with increasing coreset size.

To show the effectivity of our method we focus on instance Artificial2D. A sample of points from this instance is shown in figure 8.15. The average silhouette coefficent for this instance and coreset sizes 427, 1616, and 6431 is given in Figure 8.16. We see that even the smallest coreset suffices to approximate the coefficient quite well. The only problem is that the silhouette coefficient increases slightly with $k$. A reason for this may be that the number of centers is already relatively large compared to the number of coreset points. If some centers contain only one point, then they have silhouette coefficient exactly 1 and this may lead to slightly increasing coefficient, if $k$ is large compared to the coreset size. For our applications a coreset of size roughly 1600 will definitly suffice. There is almost no difference to one with more than 6000 points.

The highest silhouette coefficient value was achieved for 14 clusters (using the larger coreset) by the cluster centers shown in Figure 8.15. The reason why only 14 clusters were found (al-

though we had 20 cluster centers) can be explained by the fact that some of the clusters were very close to each other and so the clustering coefficient is higher when one assigns only one center to these clusters.

| Instance | Coreset | Time | Silhouette |
|---|---|---|---|
| Tower | 404 | 7.99 | 0.84 |
| | 1607 | 19.24 | 6.43 |
| Clegg | 423 | 4.69 | 0.8 |
| | 1720 | 15.07 | 6.58 |
| Monarch | 428 | 4.80 | 0.77 |
| | 1626 | 15.37 | 6.11 |
| Artificial2D | 427 | 2.52 | 0.62 |
| | 1616 | 7.73 | 4.3 |
| | 6431 | 51.89 | 45.57 |
| Artificial10D | 400 | 43.34 | 1.88 |
| | 1711 | 123.38 | 17.68 |
| Artificial20D | 408 | 139.58 | 4.18 |
| | 1778 | 442.5 | 40.62 |

Table 8.2: Time to compute clusterings and approximate average silhouette coefficients. The second column contains the overall running time (including setup). The third column gives the time required to compute the approximate average silhouette coefficient.

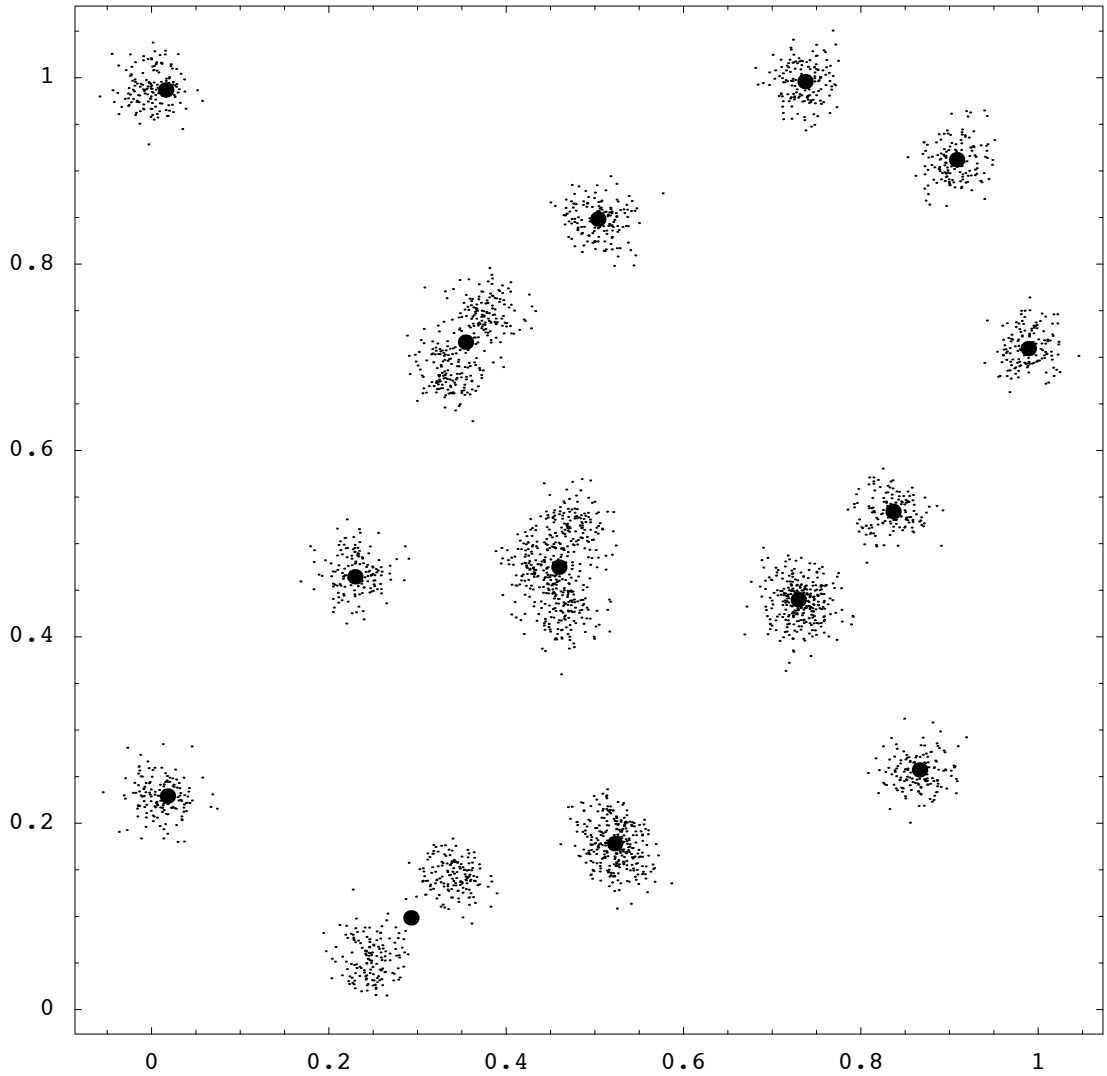


Figure 8.15: A sample of points from instance Artificial2D. The bold points are the centers that achieve the best average silhouette coefficient.

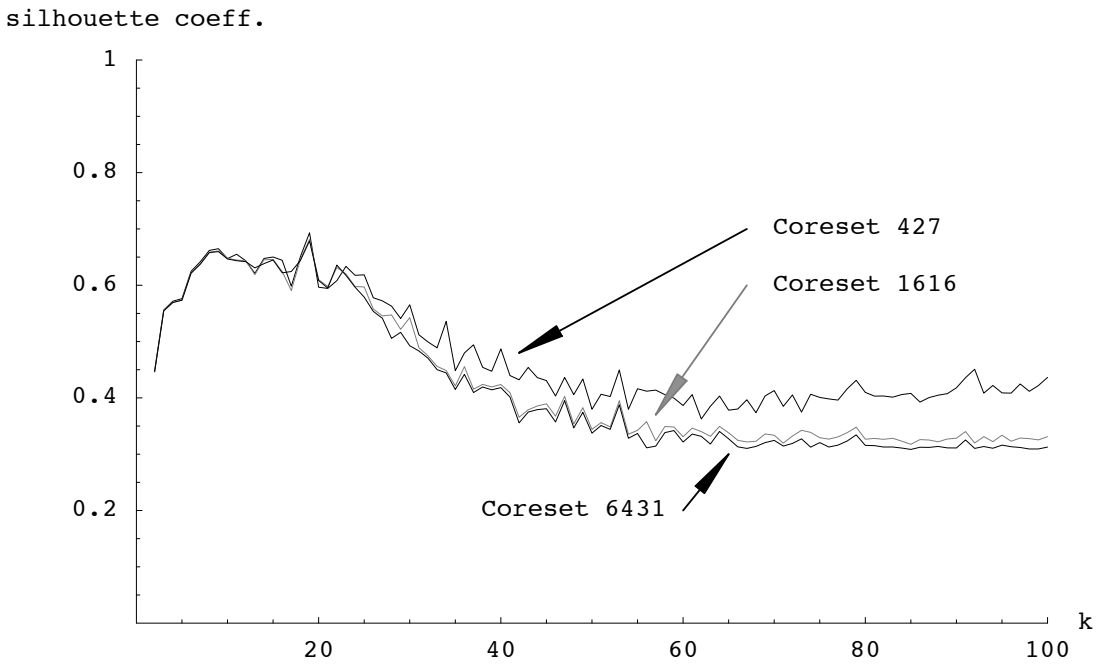


Figure 8.16: The average silhouette coefficient of Artificial2D.

### 8.3.4 Summary

Summarizing we can say that our algorithm CoreMeans performs very well compared to KMHybrid [104] for small dimension and small to medium $k$. When we compare the computation time of CoreMeans with KMHybrid we see that for one clustering the running time of both algorithms is typically dominated by the setup time. The quality of the solutions varies less than that of KMHybrid, which implies that we need fewer runs to guarantee a good solution.

The main strength of our algorithm is to quickly find relatively good approximations for many values of $k$, for example when a good value for $k$ is not known in advance. In this case, we can also use the coresets to compute the average clustering coefficient and thus to find a good choice of $k$.

# 9 Counting Motifs in Data Streams

In this chapter we present estimators for the number of triangles, the number of cliques of any size, and the number of bipartite cliques $K_{3,3}$ with three nodes in each partition for graphs given as a data stream of edges. Our data stream algorithms compute a $(1+\epsilon)$-approximation of the respective value with probability $1-\delta$.

To estimate the number of triangles for a graph given as an adjacency stream, our algorithm uses $O(\frac{1}{\epsilon^2}\cdot\log(\frac{1}{\delta})\cdot(1+\frac{|T_1|+|T_2|}{|T_3|})\cdot\log(|V|))$ memory bits, where $T_i$ denotes the set of node-triples having $i$ edges in the induced subgraph (see Definition 9.1.1). This is always better than the naive sampling algorithm that requires $O(\frac{1}{\epsilon^2}\log(\frac{1}{\delta})(1+\frac{|T_0|+|T_1|+|T_2|}{|T_3|})\cdot\log(|V|))$ memory bits, while it strongly improves the $O(\frac{1}{\epsilon^2}\cdot\log(\frac{1}{\delta})\cdot(1+\frac{|T_1|+|T_2|}{|T_3|})^3\cdot\log|V|)$ solution provided in [117]. Comparing our results in this model with the previous work in [81], we obtain a one-pass algorithm that achieves the same space bound and better update time as the three pass algorithm from [81]. The two other algorithms in [81] either require bounded maximum degree or are incomparable to our result because the space complexity depends on different parameters (e.g., the number of cycles of length 4 and 6 in the graph).

The number of memory bits used by our algorithm still depends on the value of $|T_1|/|T_3|$, that can be as large as $O(|E|\cdot|V|)$. Our method in the case of graphs in arbitrary order is therefore of practical interest for networks with a large number of triangles.

We then develop a method of greater practical relevance for graphs given as an incidence stream of edges which uses $O(\frac{1}{\epsilon^2}\log(\frac{1}{\delta})\log^2(|V|)(1+\frac{|T_2|}{|T_3|}))$ bits. To give a flavor of the quality of our result, observe that $\frac{|T_3|}{|T_2|}$ is exactly equal to $1/3$ of the inverse of the *transitivity coefficient* of the graph, a universal measure closely related to the clustering coefficient, whose value for networks of practical interest is rarely bigger than $10^5$. Our algorithmic results improve the result of [117] that requires $O(\frac{1}{\epsilon^2}\log(\frac{1}{\delta})\log(|V|)(1+\frac{|T_2|}{|T_3|})^2+d\log|V|)$ memory bits (where $d$ denotes the maximum degree of the graph) and improves over the naive sampling method.

Our method is suitable to be adapted to several other classes of subgraphs. As an example we provide an algorithm to estimate the number of cliques of size $\alpha$ in the incidence stream model that uses $O(\frac{1}{\epsilon^2}\log(\frac{1}{\delta})\log^2(|V|)(1+\frac{|S_\alpha|}{|K_\alpha|}))$ memory bits, where $S_\alpha$ is the set of stars of $\alpha$ nodes and $K_\alpha$ is the set of cliques of size $\alpha$ in the graph.

Denote by $K_{i,j}$ the set of complete bipartite cliques in the graph where each of $i$ vertices link to all of $j$ other vertices. As a last contribution we provide a data stream algorithm that provides an approximation of the number of $K_{3,3}$ of the graph in the incidence stream model ordered by destination nodes with outdegree bounded by $\Delta$ which needs $O\left(\log^2(|V|)\cdot\frac{|K_{3,1}|\cdot\Delta^2\ln(\frac{1}{\delta})}{|K_{3,3}|\cdot\epsilon^2}\right)$ mem-

ory bits.

In Section 9.1 we present our algorithm to count the number of triangles in the adjacency stream model (as defined in Section 2.2). For input graphs given as an incidence stream (see Section 2.2), we develop a better algorithm to count the number of triangles in Section 9.2. This algorithm will be generalized to cliques of any size in Section 9.3. Section 9.4 presents an algorithm to count bipartite cliques $K_{3,3}$ in the incidence stream model.

## 9.1 Counting Triangles in Adjacency Streams

We consider an undirected graph $G = (V, E)$ without self-loops. Each edge is an unordered pair of nodes $(v, w)$ such that $(v, w) = (w, v)$. We assume that $V = \{1, \ldots, n\}$ and $n$ is known in advance, and that $G$ is then given as an adjacency stream consisting of all edges in the graph as defined in Section 2.2. The edges appear in arbitrary order and no edge is repeated in the stream. There is no bound on the degree of the nodes.

**Definition 9.1.1 (Node triples, $T_0, T_1, T_2, T_3$)** *We define a* node triple *as a set $\{v_1, v_2, v_3\} \subset V$ consisting of exactly 3 different nodes of $V$. We partition the set of node triples into four sets $T_0, T_1, T_2$, and $T_3$. A node triple $\{v_1, v_2, v_3\}$ belongs to*

- $T_0$ *iff no edge exists between the nodes $v_1, v_2$, and $v_3$,*
- $T_1$ *iff exactly one of the edges $(v_1, v_2)$, $(v_2, v_3)$, and $(v_3, v_1)$ exists,*
- $T_2$ *iff exactly two of the edges $(v_1, v_2)$, $(v_2, v_3)$, and $(v_3, v_1)$ exist,*
- $T_3$ *iff all of the edges $(v_1, v_2)$, $(v_2, v_3)$, and $(v_3, v_1)$ exist (i.e. iff $\{v_1, v_2, v_3\}$ is a triangle).*

*Therefore $|T_3|$ denotes the number of triangles in $G$.*

The algorithms we present here find a $(1 \pm \epsilon)$-approximation of $|T_3|$ using local memory of size $\Theta\left(\frac{1}{\epsilon^2}\left(1 + \frac{|T_1|+|T_2|}{|T_3|}\right) \cdot \log|V|\right)$. In social graphs and the webgraph the value $(|T_1|+|T_2|)/|T_3|$ usually is $O(|V|)$.

### 9.1.1 3 Pass Algorithm

We will first present an algorithm which passes three times over the stream and computes a $(1 \pm \epsilon)$-approximation on the number of triangles. A different algorithm with the same space complexity has been presented in [81]. However, our algorithm has a significantly improved update time and as we later show, we can combine the three passes to a one-pass algorithm.

We introduce a streaming algorithm SAMPLETRIANGLE, which outputs a $\{0, 1\}$ variable $\beta$ with expected value $3|T_3|/(|T_1| + 2 \cdot |T_2| + 3 \cdot |T_3|)$. The algorithm is given in Figure 9.1.

It is easy to see that each of the passes can be implemented in a single pass over the set of edges (i.e., the input stream) using $O(1)$ memory cells, each storing $O(\log|V|)$ bits.

SAMPLETRIANGLE
**1st. Pass:**
Count the number of edges $|E|$ in the stream
**2nd. Pass:**
Sample an edge $e = (a, b)$ uniformly chosen from $E$
Choose a node $v$ uniformly from $V \setminus \{a, b\}$.
**3rd. Pass:**
**if** $(a, v) \in E \wedge (b, v) \in E$ **then** set $\beta \leftarrow 1$
**else** set $\beta \leftarrow 0$
**return** $\beta$

Figure 9.1: The 3-pass algorithm SAMPLETRIANGLE for adjacency streams

**Lemma 9.1.2** *Algorithm* SAMPLETRIANGLE *outputs a value* $\beta$ *with expected value*

$$\mathbf{E}[\beta] = \frac{3|T_3|}{|T_1| + 2 \cdot |T_2| + 3 \cdot |T_3|}$$

*Furthermore*

$$|T_1| + 2 \cdot |T_2| + 3 \cdot |T_3| = |E| \cdot (|V| - 2)$$

*and*

$$|T_3| = \mathbf{E}[\beta] \cdot |E| \cdot (|V| - 2)/3 \ .$$

**Proof :** We look at all node triples. Each triple belongs to one of the sets $T_0$, $T_1$, $T_2$, or $T_3$. The algorithm chooses such a triple by choosing an edge $e = (a, b)$ together with one node $v \in V \setminus \{a, b\}$. Therefore, no triple from $T_0$ is chosen.

We will show how many different choices of one edge and one node choose a node triple belonging to $T_1$ (resp. $T_2$, $T_3$). Since all pairs of one edge and one node have the sample probability to be chosen, we can then compute the probability to select a triangle.

Let us denote by $t = \{w_1, w_2, w_3\}$ a fixed triple from $T_1$. Wlog. let be $(w_1, w_2) \in E$ and so $(w_2, w_3), (w_3, w_1) \notin E$. The algorithm chooses $t$, iff it samples edge $(w_1, w_2)$ and vertex $w_3$. Therefore there are exactly $|T_1|$ choices of an edge and a node that select a triple from $T_1$.

Now assume $t \in T_2$. Then $t$ is chosen by SAMPLETRIANGLE, iff one of the two edges in the triple is sampled and $v$ equals to the remaining node of the triple. Therefore there are exactly $2 \cdot |T_2|$ choices of an edge and a node that select a triple from $T_2$.

For the same reason, a triple in $T_3$ is chosen whenever one of its three edges and the remaining vertex is chosen. Therefore there are exactly $3 \cdot |T_3|$ choices of an edge and a node that select a triple from $T_3$. These are exactly the choices that lead to $\beta = 1$.

We conclude that

$$\mathbf{E}[\beta] = \frac{3|T_3|}{|T_1| + 2 \cdot |T_2| + 3 \cdot |T_3|} \ .$$

Since there are $|E| \cdot (|V| - 2) = |T_1| + 2 \cdot |T_2| + 3 \cdot |T_3|$ choices to sample an edge and a node, it follows that $|T_3| = \mathbf{E}[\beta] \cdot |E| \cdot (|V| - 2)/3$. □

A streaming algorithm CountTriangles, which outputs an estimate of $|T_3|$, easily follows. It can be adjusted using an input parameter $s$.

CountTriangles ( $s \in \mathbb{N}$ )
Run $s$ instances of SampleTriangle in parallel.
Let $\beta_i$ be the value returned by the $i$th instance.
$\widetilde{T}_3 \leftarrow \left(\frac{1}{s}\sum_{i=1}^{s} \beta_i\right) \cdot |E| \cdot (|V| - 2)/3$.
**return** $\widetilde{T}_3$.

Figure 9.2: The 3-pass algorithm CountTriangles for adjacency streams

**Lemma 9.1.3** *Algorithm* CountTriangles *outputs a value* $\widetilde{T}_3$ *having expected value* $\mathbf{E}[\widetilde{T}_3] = |T_3|$. *If* $s \geq \frac{1}{\epsilon^2} \cdot \frac{|T_1|+2\cdot|T_2|+3\cdot|T_3|}{|T_3|} \cdot \ln(\frac{2}{\delta})$. *then with probability* $1 - \delta$ *the algorithm outputs a value* $\widetilde{T}_3$ *satisfying*

$$(1 - \epsilon) \cdot |T_3| \leq \widetilde{T}_3 \leq (1 + \epsilon) \cdot |T_3| \ .$$

**Proof :** We use Chernoff's Bounds [58]:

$$\mathbf{Pr}\Big[\frac{1}{s}\sum_{i=1}^{s} \beta_i \geq (1 + \epsilon)\mathbf{E}[\beta]\Big] \ < \ e^{-\epsilon^2 \cdot \mathbf{E}[\beta] \cdot s/3}$$

$$\mathbf{Pr}\Big[\frac{1}{s}\sum_{i=1}^{s} \beta_i \leq (1 - \epsilon)\mathbf{E}[\beta]\Big] \ < \ e^{-\epsilon^2 \cdot \mathbf{E}[\beta] \cdot s/2}$$

For $s \geq \frac{1}{\epsilon^2} \cdot \frac{|T_1|+2\cdot|T_2|+3\cdot|T_3|}{|T_3|} \cdot \ln(\frac{2}{\delta})$ the sum of both probabilities is bounded by $\delta$. The lemma follows now from Lemma 9.1.2 stating that $|T_3| = \mathbf{E}[\beta] \cdot |E| \cdot (|V| - 2)/3$. □

Lemma 9.1.3 guarantees that algorithm CountTriangles outputs a value $\widetilde{T}_3$ having the right expectation.

However, there could be applications which need guaranteed approximation bounds. When we run CountTriangles with a predefined number of instances $s$, it outputs an estimation $\widetilde{T}_3$ on $|T_3|$, but the requirement $s \geq \frac{1}{\epsilon^2} \cdot \frac{|T_1|+2\cdot|T_2|+3\cdot|T_3|}{|T_3|} \cdot \ln(\frac{2}{\delta})$ of Lemma 9.1.3 is impossible to check. We can not find out if we used enough instances to ensure a $(1 \pm \epsilon)$-approximation.

To overcome this problem we later develop a new algorithm CountTrianglesSafe based on CountTriangles. It outputs a pair $(\widetilde{T}_3, \tilde{\epsilon})$ and guarantees that $\widetilde{T}_3$ is a $(1 \pm \tilde{\epsilon})$-approximation of $|T_3|$ with probability $1 - \delta$.

CountTrianglesSafe ( $s \in \mathbb{N}, \psi \in (0,1)$ )
Set $r \leftarrow \lceil 2 \cdot \log(2/\psi) \rceil$.
Set $t \leftarrow \lceil s/(4\log(2/\psi)) \rceil$.
Run $r$ instances of CountTriangles($t$) in parallel.
Let $\widetilde{T_3}^{(i)}$ be the value returned by the $i$th instance.
Set $\widetilde{T_3} \leftarrow \text{median}_i(\widetilde{T_3}^{(i)})$.
Set $\tilde{\varepsilon} \leftarrow \sqrt{\frac{176}{\widetilde{T_3}} \cdot \frac{|T_1|+2\cdot|T_2|+3\cdot|T_3|}{s} \cdot \log\frac{2}{\psi}} = \sqrt{\frac{176}{\widetilde{T_3}} \cdot \frac{|E|\cdot(|V|-2)}{s} \cdot \log\frac{2}{\psi}}$.
**return** $(\widetilde{T_3}, \tilde{\varepsilon})$.

Figure 9.3: The 3-pass algorithm CountTrianglesSafe for adjacency streams

Let us first analyse the update time of CountTriangles. As before we analyse time in the Real RAM model. Note that we could also show our results in a RAM model, assuming that all stored values (each needing $O(\log|V|)$ memory bits to be stored) fit into one single register.

If we implement the different instances of our algorithm independently of each other, we require $O(s)$ time to process each edge during the third pass. We show how to reduce this to expected constant time. Before we invoke the third pass, we collect all edge-vertex pairs chosen by different instances of the algorithm. For each pair with edge $e = (a,b)$ and vertex $v$ we would like to find out whether $(a,v)$ and $(b,v)$ are in $E$. Therefore, we construct a set $M$ of *missing edges* that for each such edge-vertex pair contains the edges $(a,v)$ and $(b,v)$. Next, we construct a hash table for $M$ using a uniform hash function that requires linear space, as proposed in [107]. Now we can implement the third pass in the following way. For each edge $e$, we lookup whether it is in the set $M$. If $e \in M$ we mark it. These steps can both be done in expected constant time. In a postprocessing step we can then determine the edge-vertex pairs that are triangles.

We now develop an algorithm CountTrianglesSafe based on CountTriangles. It has a number $s$ and a desired error probability $\psi$ as input parameters and outputs a pair $(\widetilde{T_3}, \tilde{\varepsilon})$. The algorithm gives the guarantee that $\widetilde{T_3}$ is a $(1 \pm \tilde{\varepsilon})$-approximation of $|T_3|$ with probability $1-\psi$. It runs at most $s$ instances of SampleTriangle in parallel. We show the pseudocode of CountTrianglesSafe in Figure 9.3.

**Lemma 9.1.4** *Let* $(\widetilde{T_3}, \tilde{\varepsilon})$ *be the output of algorithm* CountTrianglesSafe*. With probability* $1-\psi$ *the following statements are true:*

- $(1-\tilde{\varepsilon}) \cdot |T_3| < \widetilde{T_3} < (1+\tilde{\varepsilon}) \cdot |T_3|$
- *If* $s \geq \frac{352}{\varepsilon^2} \cdot \frac{|T_1|+2\cdot|T_2|+3\cdot|T_3|}{|T_3|} \cdot \log(2/\psi)$*, then the algorithm outputs* $(\widetilde{T_3}, \tilde{\varepsilon})$ *with* $\tilde{\varepsilon} \leq \varepsilon$.

**Proof :** CountTrianglesSafe outputs $(\widetilde{T_3}, \tilde{\varepsilon})$ with $\tilde{\varepsilon} = \sqrt{\frac{176}{\widetilde{T_3}} \cdot \frac{|T_1|+2\cdot|T_2|+3\cdot|T_3|}{s} \cdot \log\frac{2}{\psi}}$.

Therefore we have $\widetilde{T_3} = \frac{176}{\tilde{\varepsilon}^2} \cdot \frac{|T_1|+2\cdot|T_2|+3\cdot|T_3|}{s} \cdot \log\frac{2}{\psi}$. Because of the choice of $t$ it follows

$$\widetilde{T_3} \geq \frac{44}{\tilde{\varepsilon}^2} \cdot \frac{|T_1| + 2 \cdot |T_2| + 3 \cdot |T_3|}{t} \ . \tag{9.1}$$

From Markov's inequality it follows that

$$\forall_{i\in\{1,\ldots,r\}} \mathbf{Pr}\big[\widetilde{T_3}^{(i)} \geq 11 \cdot \mathbf{E}[\widetilde{T_3}^{(i)}]\big] \leq \frac{1}{11} \ .$$

Since for all $i$ we have $\mathbf{E}[\widetilde{T_3}^{(i)}] = |T_3|$ it follows:

$$\forall_{i\in\{1,\ldots,r\}} \mathbf{Pr}\big[\widetilde{T_3}^{(i)} \geq 11 \cdot |T_3|\big] \leq \frac{1}{11} \ .$$

Because of $\widetilde{T_3} = \text{median}_i(\widetilde{T_3}^{(i)})$ we can only have $\widetilde{T_3} \geq 11 \cdot |T_3|$ if for at least $r/2$ values of $i$ we have $\widetilde{T_3}^{(i)} \geq 11 \cdot |T_3|$. For each single value of $i$ the probability for that is smaller than $1/11$ as shown above. The probability to have at least $r/2$ values of $i$ fulfilling that equation is therefore bounded by:

$$\begin{aligned}
&\mathbf{Pr}\big[\widetilde{T_3} \geq 11 \cdot |T_3|\big] \leq \binom{r}{r/2} \cdot \left(\frac{1}{11}\right)^{r/2} \\
&\leq \left(\frac{e \cdot r}{r/2}\right)^{r/2} \cdot \left(\frac{1}{11}\right)^{r/2} = \left(\frac{2e}{11}\right)^{r/2} \leq \left(\frac{1}{2}\right)^{r/2} \leq \psi/2 \ .
\end{aligned}$$

From inequality (9.1) we conclude

$$\mathbf{Pr}\big[\frac{44}{\tilde{\varepsilon}^2} \cdot \frac{|T_1| + 2 \cdot |T_2| + 3 \cdot |T_3|}{t} \geq 11 \cdot |T_3|\big] \leq \psi/2$$

and finally

$$\mathbf{Pr}\big[|T_3| \leq \frac{4}{\tilde{\varepsilon}^2} \cdot \frac{|T_1| + 2 \cdot |T_2| + 3 \cdot |T_3|}{t}\big] \leq \psi/2 \ .$$

In the following we therefore condition on the event that $|T_3| > \frac{4}{\tilde{\varepsilon}^2} \cdot \frac{|T_1|+2\cdot|T_2|+3\cdot|T_3|}{t}$ (which happens with probability at least $1 - \psi/2$). Then for $\delta := 1/11$ we have $\ln(2/\delta) \leq 4$ and

$$t \geq \frac{1}{\tilde{\varepsilon}^2} \cdot \frac{|T_1| + 2 \cdot |T_2| + 3 \cdot |T_3|}{|T_3|} \cdot \ln\left(\frac{2}{\delta}\right) \ .$$

Therefore by Lemma 9.1.3 each instance of CountTriangles outputs a value $\widetilde{T_3}^{(i)} \in (1 \pm \tilde{\varepsilon}) \cdot |T_3|$ with probability at least $1 - \delta = 1 - \frac{1}{11}$.

Since $\widetilde{T_3}$ is set to the median of all $\widetilde{T_3}^{(i)}$, we can only have $\widetilde{T_3} \notin (1 \pm \tilde{\varepsilon}) \cdot |T_3|$ if for at least $r/2$ of the $\widetilde{T_3}^{(i)}$ we have $\widetilde{T_3}^{(i)} \notin (1 \pm \tilde{\varepsilon}) \cdot |T_3|$. The probability for that is bounded by:

$$\begin{aligned}
&\mathbf{Pr}\big[\widetilde{T_3} \notin (1 \pm \tilde{\varepsilon}) \cdot |T_3|\big] \leq \binom{r}{r/2} \cdot \left(\frac{1}{11}\right)^{r/2} \\
&\leq \left(\frac{e \cdot r}{r/2}\right)^{r/2} \cdot \left(\frac{1}{11}\right)^{r/2} = \left(\frac{2e}{11}\right)^{r/2} \leq \left(\frac{1}{2}\right)^{r/2} \leq \psi/2 \ .
\end{aligned}$$

The first statement of the lemma follows directly.

We now show the second statement. We condition on the event that $\widetilde{T}_3 \leq (1+\tilde{\epsilon})|T_3|$, which happens with probability $1-\psi$ by the first statement of the lemma.
If $s \geq \frac{352}{\epsilon^2} \cdot \frac{|T_1|+2\cdot|T_2|+3\cdot|T_3|}{|T_3|} \cdot \log\frac{2}{\psi}$ then we have

$$|T_3| \geq \frac{352}{\epsilon^2} \cdot \frac{|T_1|+2\cdot|T_2|+3\cdot|T_3|}{s} \cdot \log\frac{2}{\psi}$$

and therefore

$$\widetilde{T}_3 \geq \frac{|T_3|}{2} \geq \frac{176}{\epsilon^2} \cdot \frac{|T_1|+2\cdot|T_2|+3\cdot|T_3|}{s} \cdot \log\frac{2}{\psi} \ .$$

It follows directly:

$$\epsilon \geq \sqrt{\frac{176}{\widetilde{T}_3} \cdot \frac{|T_1|+2\cdot|T_2|+3\cdot|T_3|}{s} \cdot \log\frac{2}{\psi}} = \tilde{\epsilon} \ .$$

□

Notice that $r \cdot t \leq s$ for $s \geq 8\log(2/\psi)$. Therefore we have at most $s$ instances of SAMPLETRIANGLE running in parallel. The time SAMPLETRIANGLESAFE(s) needs to process an edge in the stream is therefore at most the time SAMPLETRIANGLE(s) needs.

We summarize our results in the following theorem. We remark that we significantly improve the update time over the previously best result from [81] while achieving the same space complexity. The update time in [81] is roughly proportional to the space complexity compared to expected constant time for our algorithm.

**Theorem 24** *There is a 3-Pass streaming algorithm to count the number of triangles in a stream of edges up to a multiplicative error of* $1 \pm \epsilon$ *with probability at least* $1-\psi$*, which needs* $O(\frac{1}{\epsilon^2} \cdot \log(\frac{1}{\psi}) \cdot (1 + \frac{|T_1|+|T_2|}{|T_3|}) \cdot \log|V|)$ *memory bits and constant expected update time.* □

## 9.1.2 1 Pass Algorithm

In this section we show that the previous 3-pass algorithms can be implemented in one pass using the same amount of space and constant expected amortized update time, if $|E|$ is significantly larger than the number of instances we run.

We first show how to adapt algorithm SAMPLETRIANGLE. We observe that we can find a random edge in one pass by reservoir sampling [120], i.e. choosing the first edge as a sample edge and replacing this edge by the $i$th edge of the stream with probability $1/i$. It is known that this method can be implemented in $O(\log|V|)$ expected time per sample (not counting the time to read the stream) by randomly choosing the next index of the replacing edge according to an appropriate probability distribution.

```
SAMPLETRIANGLEONEPASS
  i ← 1
  for each edge e = (u, w) in the stream do
    Flip a coin. With probability 1/i do
      a ← u; b ← w;
      v ← Node uniformly chosen from V \ {a, b}
      x ← false; y ← false
    end do
    if e = (a, v) then x ← true
    if e = (b, v) then y ← true
    i ← i + 1
  end for
  if x = true ∧ y = true then return β ← 1 else return β ← 0.
```

Figure 9.4: The 1-pass algorithm SAMPLETRIANGLEONEPASS for adjacency streams

We combine this with the third pass and obtain algorithm SAMPLETRIANGLEONEPASS as shown in Figure 9.4. It may happen that we sample an edge $e = (a, b)$ of the stream together with a node $v$, but we do not see the edge $(a, v)$ or $(b, v)$ in the subsequent stream (because they appeared before the edge $e$). In this case, we do not detect $a, b, v$ as a triangle. However, we detect $a, b, v$, iff $(a, b)$ is the first edge of the triangle that appears in the stream. This changes the expected value of $\beta$ by a factor of 3.

**Lemma 9.1.5** *Algorithm* SAMPLETRIANGLEONEPASS *outputs a value* $\beta$ *having expected value*

$$\mathbf{E}[\beta] = \frac{|T_3|}{|T_1| + 2 \cdot |T_2| + 3 \cdot |T_3|} \ .$$

**Proof :** The proof is similar to the proof of Lemma 9.1.2, taking into account that only 1/3 of the choices that select a $T_3$-triple actually detect a triangle and lead to value $\beta = 1$. □

An algorithm COUNTTRIANGLESONEPASS outputting an estimation $\widetilde{T}_3$ of $|T_3|$ can be developed similarly to COUNTTRIANGLES (see Figure 9.5).

**Lemma 9.1.6** *Algorithm* COUNTTRIANGLESONEPASS *outputs a value* $\widetilde{T}_3$ *having expected value* $\mathbf{E}[\widetilde{T}_3] = |T_3|$. *If* $s \geq \frac{3}{\epsilon^2} \cdot \frac{|T_1|+2\cdot|T_2|+3\cdot|T_3|}{|T_3|} \cdot \ln(\frac{2}{\delta})$. *then with probability* $1 - \delta$ *the algorithm outputs a value* $\widetilde{T}_3$ *satisfying*

$$(1 - \epsilon) \cdot |T_3| \leq \widetilde{T}_3 \leq (1 + \epsilon) \cdot |T_3| \ .$$

**Proof :** The proof is similar to the proof of Lemma 9.1.3 (with an additional factor of 3 in most formulas). □

COUNTTRIANGLESONEPASS ( $s \in \mathbb{N}$ )
Run $s$ instances of SAMPLETRIANGLEONEPASS in parallel.
Let $\beta_i$ be the value returned by the $i$th instance.
$\widetilde{T}_3 \leftarrow \left(\frac{1}{s}\sum_{i=1}^{s} \beta_i\right) \cdot |E| \cdot (|V| - 2)$.
**return** $\widetilde{T}_3$.

Figure 9.5: The 1-pass algorithm COUNTTRIANGLESONEPASS for adjacency streams

COUNTTRIANGLESONEPASSSAFE ( $s \in \mathbb{N}, \psi \in (0,1)$ )
Set $r \leftarrow \lceil 2 \cdot \log(2/\psi) \rceil$.
Set $t \leftarrow \lceil s/(4\log(2/\psi)) \rceil$.
Run $r$ instances of COUNTTRIANGLESONEPASS$(t)$ in parallel.
Let $\widetilde{T}_3^{(i)}$ be the value returned by the $i$th instance.
$\widetilde{T}_3 \leftarrow \text{median}_i(\widetilde{T}_3^{(i)})$.
Set $\tilde{\epsilon} \leftarrow \sqrt{\frac{528}{\widetilde{T}_3} \cdot \frac{|T_1|+2\cdot|T_2|+3\cdot|T_3|}{s} \cdot \log\frac{2}{\psi}} = \sqrt{\frac{528}{\widetilde{T}_3} \cdot \frac{|E|\cdot(|V|-2)}{s} \cdot \log\frac{2}{\psi}}$.
**return** $(\widetilde{T}_3, \tilde{\epsilon})$.

Figure 9.6: The 1-pass algorithm COUNTTRIANGLESONEPASSSAFE for adjacency streams

Similarly to COUNTTRIANGLESSAFE we can develop a one pass algorithm COUNTTRIANGLESONEPASSSAFE (Figure 9.6) which outputs an approximation $\widetilde{T}_3$ together with an approximation guarantee.

**Lemma 9.1.7** *Let* $(\widetilde{T}_3, \tilde{\epsilon})$ *be the output of algorithm* COUNTTRIANGLESONEPASSSAFE. *With probability* $1 - \psi$ *the following statements are true:*

- $(1 - \tilde{\epsilon}) \cdot |T_3| < \widetilde{T}_3 < (1 + \tilde{\epsilon}) \cdot |T_3|$
- *If* $s \geq \frac{1056}{\epsilon^2} \cdot \frac{|T_1|+2\cdot|T_2|+3\cdot|T_3|}{|T_3|} \cdot \log(2/\psi)$, *then the algorithm outputs* $(\widetilde{T}_3, \tilde{\epsilon})$ *with* $\tilde{\epsilon} \leq \epsilon$.

**Proof :** The proof can be done similarly to the proof of Lemma 9.1.3 (with an additional factor of 3 in most formulas). □

By applying the reservoir sampling algorithm from [120] to select the edge, the selection requires $O(\log |V|)$ expected time for each instance of SAMPLETRIANGLEONEPASS for the whole stream. Additionally we use the hash table approach from the previous chapter to efficiently find instances of SAMPLETRIANGLEONEPASS which search for an edge in the stream. Alltogether we get expected $O(1 + s \cdot \frac{\log |E|}{|E|})$ update time per edge in the stream.

**Theorem 25** *There is a 1-Pass streaming algorithm to count the number of triangles in a stream of edges up to a multiplicative error of* $1 \pm \epsilon$ *with probability at least* $1 - \psi$*, which needs* $O(\frac{1}{\epsilon^2} \cdot \log(\frac{1}{\psi}) \cdot (1 + \frac{|T_1|+|T_2|}{|T_3|}) \cdot \log|V|)$ *memory bits and expected update time* $O(1 + \frac{1}{\epsilon^2} \cdot \frac{|T_1|+|T_2|}{|T_3|} \cdot \frac{\log|E|}{|E|} \cdot \log\frac{1}{\psi})$.

## 9.2 Counting Triangles in Incidence Streams

When a graph $G = (V, E)$ is coded as an incidence stream (see Section 2.2) all edges incident to the same vertex appear subsequently in the stream. First arrive all edges incident to vertex $v_1$, followed by all edges incident to $v_2$, and so on. The ordering $v_1, \ldots, v_n$ of the vertices can be arbitrarily, i.e. determined by an adversary. We consider undirected graphs and so each edge appears twice (within the incidence list of both incident nodes). There is no bound on the degree of the nodes (in contrast to [117]).

Often large graphs (e.g. the webgraph) are stored on hard discs as incidence lists of edges. Our methods can therefore be used to approximate the number of triangles in these graphs using only sequential access to the data.

### 9.2.1 3 Pass Algorithm

We again will first develop a 3-pass algorithm, and later combine the passes to get a one pass algorithm. Let $d_i$ denote the degree of node $v_i$. The 3-pass algorithm SAMPLETRIANGLE2 is presented in Figure 9.7. The algorithm SAMPLETRIANGLE2 can be implemented using $O(1)$ memory cells, each consisting of $O(\log|V|)$ bits.

SAMPLETRIANGLE2
**1st. Pass:**
Count the number $P$ of paths of length 2 in the graph $G$.
**2nd. Pass:**
Uniformly choose one of these paths using algorithm UNIFORMTWOPATH($P$).
Let $(a, v, b)$ be this path.
**3rd. Pass:**
Test if edge $(a, b)$ appears within the stream.
**if** $(a, b) \in E$ **then** set $\beta \leftarrow 1$
**else** set $\beta \leftarrow 0$
**return** $\beta$

Figure 9.7: The 3-pass algorithm SAMPLETRIANGLE2 for incidence streams

We observe that the number of paths of length 2 in the graph G is exactly

$$P := |T_2| + 3 \cdot |T_3| = \sum_{i=1}^{|V|} d_i \cdot (d_i - 1)/2 \ .$$

Thus we can easily count the number of paths of length 2 by determining the degree of each node. This is possible because the edges appear as an incidence stream.

The second pass can be implemented using reservoir sampling. However, we propose a different approach which achieves slightly better amortized running time and is based on the following idea. If $v$ is incident to the nodes $w_1, w_2, ..., w_d$, we define an order on the possible paths of length 2 with $v$ in the middle in the following way: $(w_1, v, w_2) < (w_1, v, w_3) < (w_2, v, w_3) < (w_1, v, w_4), ...$ . The triples $(w_i, v, w_j)$ are ordered firstly by $\max\{i, j\}$. Ties are ordered by $i$.

We choose a value $k \in \{1, \ldots, P\}$ uniformly at random and want to select the k-th triple $(w_i, v, w_j)$ in the order given above. $i$ and $j$ can be computed from $k$ using the formulas given in Figure 9.8. The k-th triple is chosen, if the node $v$ is in the middle of enough paths of length 2. Otherwise we search for the $k - d_v \cdot (d_v - 1)/2$-th path within the next incidence list.

The algorithm UNIFORMTWOPATH is presented in Figure 9.8.

UNIFORMTWOPATH(P)
  Select value k uniformly from the set $\{1, \ldots, P\}$
  **For each** node $v$ in the incidence list **do**
    **If** $k > 0$ **then**
      Set $j \leftarrow \left\lceil \sqrt{2k + \frac{1}{4}} + \frac{1}{2} \right\rceil$
      Set $i \leftarrow j - \frac{j^2 - j}{2} + k - 1$
      Pass over the complete incidence list of node $v$.
      **If** incidence list of $v$ contains more than $j$ edges **then**
        $a \leftarrow$ the $i$th node in the incidence list of $v$
        $b \leftarrow$ the $j$th node in the incidence list of $v$
        $w \leftarrow v$
      **end if**
      $d \leftarrow$ degree of node $v$
      $k \leftarrow k - \frac{d^2 - d}{2}$
    **end if**
  **end do**
  **return** edges $(a, w, b)$

Figure 9.8: The 1-pass algorithm UNIFORMTWOPATH for incidence streams

**Lemma 9.2.1** *Algorithm* SAMPLETRIANGLE2 *outputs a value* $\beta$ *with expected value*

$$\mathbf{E}[\beta] = \frac{3 \cdot |T_3|}{|T_2| + 3 \cdot |T_3|}$$

**Proof :** We look at all triples of nodes in $V$. Each triple belongs to one of the sets $T_0$, $T_1$, $T_2$, or $T_3$. The algorithm chooses such a triple by choosing a node $v$ together with two adjacent edges. Therefore the selected triples belong to the set $T_2 \cup T_3$. We select a triple from $T_2$, if we choose the unique node adjacent to both edges and the corresponding edges. Therefore there are exactly $|T_2|$ choices that choose a triple belonging to $T_2$.

A triple from set $T_3$ can be chosen in three different ways by selecting one of the three nodes of the triple together with both adjacent edges. Since each choice of a path of length two has the same probability, the probability of choosing a triple in $T_3$ is exactly $3 \cdot |T_3|/(|T_2| + 3 \cdot |T_3|)$ as stated. □

A streaming algorithm COUNTTRIANGLES2, which outputs an estimate of $|T_3|$, easily follows. It can be adjusted using an input parameter $s$ and is given in Figure 9.9.

COUNTTRIANGLES2 ( $s \in \mathbb{N}$ )
Run $s$ instances of SAMPLETRIANGLE2 in parallel.
Let $\beta_i$ be the value returned by the $i$th instance.
$\widetilde{T}_3 \leftarrow \left(\frac{1}{s}\sum_{i=1}^{s} \beta_i\right) \cdot \frac{|T_2|+3\cdot|T_3|}{3} = \left(\frac{1}{s}\sum_{i=1}^{s} \beta_i\right) \cdot \sum_{v\in V} d_v \cdot (d_v - 1)/6.$
**return** $\widetilde{T}_3$.

Figure 9.9: The 3-pass algorithm COUNTTRIANGLES2 for incidence streams

**Lemma 9.2.2** *Algorithm* COUNTTRIANGLES2 *outputs a value* $\widetilde{T}_3$ *having expected value* $\mathbf{E}[\widetilde{T}_3] = |T_3|$. *If* $s \geq \frac{1}{\epsilon^2} \cdot \frac{|T_2|+3\cdot|T_3|}{|T_3|} \cdot \ln(\frac{2}{\delta})$. *then with probability* $1 - \delta$ *the algorithm outputs a value* $\widetilde{T}_3$ *satisfying*

$$(1 - \epsilon) \cdot |T_3| \leq \widetilde{T}_3 \leq (1 + \epsilon) \cdot |T_3| \ .$$

**Proof :** Equivalent to the proof of Lemma 9.1.3. □

We can again develop an algorithm COUNTTRIANGLESSAFE2 based on COUNTTRIANGLES2. It has a number $s$ and a desired error probability $\psi$ as input parameters and outputs a pair $(\widetilde{T}_3, \tilde{\epsilon})$ where $\tilde{\epsilon}$ signals that $\widetilde{T}_3$ is an $(1 \pm \tilde{\epsilon})$-approximation of $|T_3|$. It uses at most $s$ parallel instances of SAMPLETRIANGLE2. We show the pseudocode of COUNTTRIANGLESSAFE2 in Figure 9.10.

**Lemma 9.2.3** *Let* $(\widetilde{T}_3, \tilde{\epsilon})$ *be the output of algorithm* COUNTTRIANGLESSAFE2. *With probability* $1 - \psi$ *the following statements are true:*

- $(1 - \tilde{\epsilon}) \cdot |T_3| < \widetilde{T}_3 < (1 + \tilde{\epsilon}) \cdot |T_3|$

COUNTTRIANGLESSAFE2 ( $s \in \mathbb{N}, \psi \in (0,1)$ )
Set $r \leftarrow \lceil 2 \cdot \log(2/\psi) \rceil$.
Set $t \leftarrow \lceil s/(4\log(2/\psi)) \rceil$.
Run $r$ instances of COUNTTRIANGLES2($t$) in parallel.
Let $\widetilde{T}_3^{(i)}$ be the value returned by the $i$th instance.
$\widetilde{T}_3 \leftarrow \text{median}_i(\widetilde{T}_3^{(i)})$.
Set $\tilde{\epsilon} \leftarrow \sqrt{\frac{176}{\widetilde{T}_3} \cdot \frac{|T_2|+3\cdot|T_3|}{s} \cdot \log\frac{2}{\psi}} = \sqrt{\frac{88}{\widetilde{T}_3} \cdot \frac{\sum_{v\in V} d_v\cdot(d_v-1)}{s} \cdot \log\frac{2}{\psi}}$.
**return** $(\widetilde{T}_3, \tilde{\epsilon})$.

Figure 9.10: The 3-pass algorithm COUNTTRIANGLESSAFE2 for adjacency streams

- *If* $s \geq \frac{352}{\epsilon^2} \cdot \frac{|T_2|+3\cdot|T_3|}{|T_3|} \cdot \log(2/\psi)$*, then the algorithm outputs* $(\widetilde{T}_3, \tilde{\epsilon})$ *with* $\tilde{\epsilon} \leq \epsilon$.

**Proof :** Equivalent to the proof of Lemma 9.1.4. □

To get small amortized expected update time we proceed as follows. Each time when the incidence list of a new vertex starts, we compute the values $i$ and $j$ for every instance. Then we insert the $j$-values into a global priority queue keeping a pointer to the corresponding instance. When we then process the incidence list of the current vertex we maintain a global counter for the number of neighbors of the current vertex we have seen. If this number is equal to the smallest value stored in the priority queue we remove it and process the corresponding instance. After the incidence list has been processed, we empty the priority queue. This way, each instance of the algorithm requires $O(1)$ time per vertex. Additionally, we need $O(s \cdot \log|V|)$ time to process the removal of the smallest element in the priority queue. Overall, the amortized cost of the second pass is $O(1 + s \cdot \frac{|V|}{|E|})$, which is constant for moderately large values of $|E|$. To implement the third pass we use hashing in a similar way as in the algorithm for adjacency lists. This leads to expected constant update time for the third pass.

**Theorem 26** *There is a 3-Pass streaming algorithm to count the number of triangles in incidence streams up to a multiplicative error of* $1 \pm \epsilon$ *with probability at least* $1 - \psi$*, which needs*

$$O\left(\frac{1}{\epsilon^2} \cdot \frac{|T_2| + 3\cdot|T_3|}{|T_3|} \cdot \log\left(\frac{1}{\psi}\right) \cdot \log|V|\right)$$

*memory bits and amortized expected update time*

$$O\left(1 + \frac{1}{\epsilon^2} \cdot \frac{|T_2| + 3\cdot|T_3|}{|T_3|} \cdot \log\left(\frac{1}{\psi}\right) \cdot \frac{|V|}{|E|}\right) \quad .$$

### 9.2.2 1 Pass Algorithm

To get a one pass algorithm we will again combine the passes of SAMPLETRIANGLE2. The first pass only counts the number $P$ of paths of length 2 in the graph. Instead of counting this number in advance, we will start an instance of the streaming algorithm for each guess $\tilde{P}$ of the number of length-2-paths in the set $\{1, 2, 4, 8, ..., |V|^3\}$. In parallel we will count $P$. At the end we can find one instance started with a value $\tilde{P}$ satisfying $P \leq \tilde{P} < 2P$. We choose the result of this instance as the result of our algorithm.

We have to develop a data stream algorithm which only relies on an estimation $\tilde{P}$ fulfilling $P \leq \tilde{P} < 2P$.

To combine the second and third pass we only test all edges seen after drawing the sample. The algorithm SAMPLETRIANGLEONEPASS2 is given in Figure 9.11 and can be implemented using $O(\log^2 |V|)$ memory bits.

SAMPLETRIANGLEONEPASS2
**do the following things in parallel for** $i = 0, 1, 2, \ldots, \lfloor \log(|V|^3) \rfloor$ **:**
Let be $\widetilde{P}_i := 2^i$.
Uniformly choose one path of length 2 using algorithm UNIFORMTWOPATH($\widetilde{P}_i$).
**If** UNIFORMTWOPATH did not select a path until the end of the stream, **then return** $\perp$.
Let $(a, v, b)$ be the selected path.
After choosing the path, test if edge $(a, b)$ appears within the rest of the stream.
**if** $(a, b)$ appears in the stream after the incidence list of $v$ **then** set $\beta_i \leftarrow 1$
**else** set $\beta_i \leftarrow 0$
**in parallel to the for loop do:** count the number $P = \sum_v d_v(d_v - 1)/2$.
set $\beta \leftarrow \beta_{\lceil \log P \rceil}$
**return** $\beta$

Figure 9.11: The 1-pass algorithm SAMPLETRIANGLEONEPASS2 for incidence streams

**Lemma 9.2.4** *Algorithm* SAMPLETRIANGLEONEPASS2 *outputs with probability at least* $1/2$ *a value* $\beta$ *having expected value*

$$\mathbf{E}[\beta] = \frac{2 \cdot |T_3|}{|T_2| + 3 \cdot |T_3|} \ .$$

*Otherwise it outputs the value* $\perp$.

**Proof :** We set $\beta = \beta_i$ with $i = \lceil \log P \rceil$ at the end of the algorithm. This value of $\beta_i$ has been set to 0 or 1, if UNIFORMTWOPATH($\widetilde{P}_i$) did select a path of length two. Since $\widetilde{P}_i = 2^{\lceil \log P \rceil}$ we have $P \leq \widetilde{P}_i < 2 \cdot P$.

UNIFORMTWOPATH($\widetilde{P}_i$) selects a path of length two by first choosing $k \in \{1, \ldots, \widetilde{P}_i\}$ uniformly at random and then selecting the $k-$th path of length two in the stream. If $k \leq \widetilde{P}_i/2$ we

have $k \leq P$ and therefore a path is selected. This happens with probability $1/2$ and in that case SAMPLETRIANGLEONEPASS2 does not output $\perp$.

Let us now condition on the event that $\beta \neq \perp$ and analyse the expected value of $\beta$.

Let $\{a, b, c\}$ be a fixed triangle. Wlog. we assume that we see the incidence list of $a$ first in the stream, then the incidence list of $b$ and then the incidence list of $c$.

In the algorithm SAMPLETRIANGLE2 we detected the triangle by selecting $(a, b, c)$, $(c, a, b)$, or $(b, c, a)$ as the path of length two.

Now only the selections of $(a, b, c)$ or $(c, a, b)$ lead to a detection of the triangle (because the edge $(c, a)$ resp. $(c, b)$ appears in the incidence stream after selecting $b$ resp. $a$). The selection of $(b, c, a)$ as a path of length two is done using the incidence list of $c$. Therefore the incidence lists of $a$ and $b$ have passed and we don't detect the edge $(a, b)$. We conclude that the probability to output $\beta = 1$ (under the condition that $\beta \neq \perp$) is exactly $2/3$ times the probability of SAMPLETRIANGLE2 to output $\beta = 1$.

□

A streaming algorithm COUNTTRIANGLESONEPASS2, which outputs an estimate of $|T_3|$, easily follows. It can be adjusted using an input parameter $s$ and is given in Figure 9.12.

COUNTTRIANGLESONEPASS2 ( $s \in \mathbb{N}$ )
Run $s$ instances of SAMPLETRIANGLEONEPASS2 in parallel.
Let $s'$ be the number of instances not returning $\perp$.
Let $\beta_i$ be the value returned by the $i$th such instance (not returning $\perp$).
$\widetilde{T}_3 \leftarrow \left(\frac{1}{s'} \sum_{i=1}^{s'} \beta_i\right) \cdot \frac{|T_2|+3\cdot|T_3|}{2} = \left(\frac{1}{s'} \sum_{i=1}^{s'} \beta_i\right) \cdot \sum_{v \in V} d_v \cdot (d_v - 1)/4.$
**return** $\widetilde{T}_3$.

Figure 9.12: The 1-pass algorithm COUNTTRIANGLESONEPASS2 for incidence streams

**Lemma 9.2.5** *Algorithm* COUNTTRIANGLESONEPASS2 *outputs a value* $\widetilde{T}_3$ *having expected value* $\mathbf{E}[\widetilde{T}_3] = |T_3|$. *If* $\epsilon \leq 1/2$ *and* $s \geq \frac{6}{\epsilon^2} \cdot \frac{|T_2|+3\cdot|T_3|}{T_3} \cdot \ln(\frac{4}{\delta})$. *then with probability* $1 - \delta$ *the algorithm outputs a value* $\widetilde{T}_3$ *satisfying*

$$(1 - \epsilon) \cdot |T_3| \leq \widetilde{T}_3 \leq (1 + \epsilon) \cdot |T_3| \ .$$

**Proof :** The expected value of $\widetilde{T}_3$ follows easily from Lemma 9.2.4.

Let be $\epsilon \leq 1/2$ and $s \geq \frac{6}{\epsilon^2} \cdot \frac{|T_2|+3\cdot|T_3|}{T_3} \cdot \ln(\frac{4}{\delta})$. First we show that $s' \geq \frac{3}{2\cdot\epsilon^2} \cdot \frac{|T_2|+3\cdot|T_3|}{T_3} \cdot \ln(\frac{4}{\delta})$ with probability at least $1 - \delta/2$.

Let $c_j \in \{0, 1\}$ be the random indicator variable which is 1 if the $j$th instance of SAMPLETRIANGLEONEPASS2 returns 0 or 1 and which is 0 if the $j$th instance returns $\perp$. By Lemma 9.2.4

we have $\mathbf{E}[c_j] \geq 1/2$. By Chernoff Bounds [58]:

$$\mathbf{Pr}\big[\sum_{j=1}^{s} c_j \leq (\frac{s}{2} \cdot \mathbf{E}[c_j])\big] = \mathbf{Pr}\big[\frac{1}{s}\sum_{j=1}^{s} c_j \leq \frac{1}{2} \cdot \mathbf{E}[c_j]\big] < e^{-\frac{1}{4}\cdot\mathbf{E}[c_j]\cdot s/2} \ .$$

Therefore we have:

$$\mathbf{Pr}\big[s' \leq \frac{3}{2\cdot\epsilon^2} \cdot \frac{|T_2| + 3\cdot|T_3|}{T_3} \cdot \ln(\frac{4}{\delta})\big] \leq e^{-s/16} \leq \delta/4$$

for $\epsilon < 1/2$.

We now condition on the event that $s' \geq \frac{3}{2\cdot\epsilon^2} \cdot \frac{|T_2|+3\cdot|T_3|}{T_3} \cdot \ln(\frac{4}{\delta})$.
We again use Chernoff Bounds [58]:

$$\mathbf{Pr}\big[\frac{1}{s'}\sum_{i=1}^{s'} \beta_i \geq (1+\epsilon)\mathbf{E}[\beta]\big] \ < \ e^{-\epsilon^2\cdot\mathbf{E}[\beta]\cdot s'/3}$$

$$\mathbf{Pr}\big[\frac{1}{s'}\sum_{i=1}^{s'} \beta_i \leq (1-\epsilon)\mathbf{E}[\beta]\big] \ < \ e^{-\epsilon^2\cdot\mathbf{E}[\beta]\cdot s'/2}$$

For $s' \geq \frac{3}{2\cdot\epsilon^2} \cdot \frac{|T_2|+3\cdot|T_3|}{|T_3|} \cdot \ln(\frac{4}{\delta})$ the sum of both probabilities is bounded by $\delta/2$.

□

We can again develop an algorithm CountTrianglesOnePassSafe2 based on CountTrianglesOnePass2. It has a number $s$ and a desired error probability $\psi$ as input parameters and outputs a pair $(\widetilde{T_3}, \tilde{\epsilon})$. $\widetilde{T_3}$ then is a $(1 \pm \tilde{\epsilon})$-approximation of $|T_3|$ with probability $1-\psi$. It uses at most $s$ parallel instances of SampleTriangleOnePass2. We show the pseudocode of CountTrianglesOnePassSafe2 in Figure 9.13.

CountTrianglesOnePassSafe2 ( $s \in \mathbb{N}, \psi \in (0,1)$ )
Set $r \leftarrow \lceil 2 \cdot \log(4/\psi) \rceil$.
Set $t \leftarrow \lceil s/(4\log(4/\psi)) \rceil$.
Run $r$ instances of CountTrianglesOnePass2($t$) in parallel.
Let $\widetilde{T_3}^{(i)}$ be the value returned by the $i$th instance.
$\widetilde{T_3} \leftarrow \text{median}_i(\widetilde{T_3}^{(i)})$.
Set $\tilde{\epsilon} \leftarrow \sqrt{\frac{1056}{\widetilde{T_3}} \cdot \frac{|T_2|+3\cdot|T_3|}{s} \cdot \log\frac{4}{\psi}} = \sqrt{\frac{528}{\widetilde{T_3}} \cdot \frac{\sum_{v\in V} d_v\cdot(d_v-1)}{s} \cdot \log\frac{4}{\psi}}$.
**return** $(\widetilde{T_3}, \tilde{\epsilon})$.

Figure 9.13: The 1-pass algorithm CountTrianglesOnePassSafe2 for adjacency streams

**Lemma 9.2.6** *Let* $(\widetilde{T}_3, \tilde{\epsilon})$ *be the output of algorithm* CountTrianglesOnePassSafe2. *With probability* $1-\psi$ *the following statements are true:*

- $(1-\tilde{\epsilon}) \cdot |T_3| < \widetilde{T}_3 < (1+\tilde{\epsilon}) \cdot |T_3|$
- *If* $s \geq \frac{2112}{\epsilon^2} \cdot \frac{|T_2|+3\cdot|T_3|}{|T_3|} \cdot \log(4/\psi)$, *then the algorithm outputs* $(\widetilde{T}_3, \tilde{\epsilon})$ *with* $\tilde{\epsilon} \leq \epsilon$.

**Proof :** Equivalent to the proof of Lemma 9.1.4. □

We use techniques to reduce the amortized update time as shown in the previous section. Since we start $O(\log|V|)$ parallel instances for different guesses of $\tilde{P}$, the amortized update time increases by a factor of $O(\log|V|)$.

**Theorem 27** *There is a 1-Pass streaming algorithm to count the number of triangles in incidence streams up to a multiplicative error of* $1 \pm \epsilon$ *with probability at least* $1-\psi$, *which needs*

$$O\left(\frac{1}{\epsilon^2} \cdot \left(1+\frac{|T_2|}{|T_3|}\right) \log\left(\frac{1}{\psi}\right) \cdot \log^2|V|\right)$$

*memory bits and amortized expected update time*

$$O\left(\log(|V|) \cdot \left(1+\frac{1}{\epsilon^2} \cdot \left(1+\frac{|T_2|}{|T_3|}\right) \log\left(\frac{1}{\psi}\right) \cdot \left(\frac{|V|}{|E|}\right)\right)\right) \quad .$$

# 9.3 Counting Cliques of Arbitrary Size

Using the approach of the previous sections we can count cliques of $\alpha$ nodes in incidence streams as well using one pass. We assume that $\alpha$ is a small constant. Let $S_\alpha$ be the set of $\alpha$-stars ($\alpha$ nodes $v_1, \ldots, v_\alpha$ and edges $(v_1, v_2), \ldots, (v_1, v_\alpha)$) in $G$ and $K_\alpha$ be the set of cliques of size $\alpha$ in $G$. Our memory bounds will depend on $|S_\alpha|/|K_\alpha|$. In network analysis we are interested in those networks where this ratio is small, for example constant.

We use the method UniformStar given in Figure 9.14 to uniformly choose an $\alpha$-star. It uses $O(\log|V|)$ memory bits and has expected running time $O(|V| \cdot \log|V|)$, not counting the time to read the stream. The selected star is then used by SampleCliqueOnePass given in Figure 9.15. Each instance of SampleCliqueOnePass uses $O(\log^2|V|)$ memory bits. When we run $s$ parallel instances of SampleCliqueOnePass, the whole method has amortized expected running time $O(s \cdot \frac{|V|\cdot\log|V|}{|E|} + 1)$ per edge.

**Lemma 9.3.1** *Algorithm* SampleCliqueOnePass *with probability at least* $1/2$ *outputs a value* $\beta$ *having expected value*

$$\mathbf{E}[\beta] = \frac{2 \cdot |K_\alpha|}{|S_\alpha|}$$

*Otherwise it outputs the value* $\perp$.

UNIFORMSTAR(P)
Select value k uniformly from the set $\{1, \ldots, P\}$.
**For each** node $v$ in the stream **do**
Use the reservoir sampling technique of [120] to obtain $\alpha - 1$ sample nodes from the incidence list of $v$.
Let d be the degree of node $v$.
**If** $\binom{d}{\alpha-1} \geq k$ **then**
return sample nodes.
**end if**
$k \leftarrow k - \binom{d}{\alpha-1}$
**end do**
**return** $\perp$.

Figure 9.14: The 1-pass algorithm UNIFORMSTAR for adjacency streams

**Proof :** We set $\beta = \beta_i$ with $i = \lceil \log P \rceil$ at the end of the algorithm. This value of $\beta_i$ has been set to 0 or 1, if UNIFORMSTAR($\widetilde{P}_i$) did select a star. Since $\widetilde{P}_i = 2^{\lceil \log P \rceil}$ we have $P \leq \widetilde{P}_i < 2 \cdot P$.

UNIFORMSTAR($\widetilde{P}_i$) selects a star by first choosing $k \in \{1, \ldots, \widetilde{P}_i\}$ uniformly at random and then selecting the k-th star in the stream. If $k \leq \widetilde{P}_i/2$ we have $k \leq P$ and therefore a star is selected. This happens with probability $1/2$ and in that case SAMPLECLIQUEONEPASS outputs $\beta = 0$ or $\beta = 1$.

Let us now condition on the event that $\beta \neq \perp$ and analyse the expected value of $\beta$.

If the star is sampled from the incidence list of $v$ and $v$ is the first or the second of the star-nodes within the stream we can find all other edges after selecting the sample star. So if the star belongs to a clique, this clique is detected.

However, if $v$ is the third or a later node in the stream we miss the edge connecting the first and second node of the star. When we fix a clique, the probability that the chosen node $v$ is first or second node of the clique in the stream is $2/\alpha$. Therefore two choices of stars lead to a detection of a fixed clique. It follows that the expected value of $\beta$ is

$$\mathbf{E}[\beta] = \frac{2}{\alpha} \cdot \frac{\alpha \cdot |K_\alpha|}{|S_\alpha|}$$

□

A streaming algorithm COUNTCLIQUESONEPASS, which outputs an estimate of $|K_\alpha|$, easily follows. It can be adjusted using an input parameter s and is given in Figure 9.16.

**Lemma 9.3.2** *Algorithm* COUNTCLIQUESONEPASS *outputs a value* $\widetilde{K_\alpha}$ *having expected value* $\mathbf{E}[\widetilde{K_\alpha}] = |K_\alpha|$.

SAMPLECLIQUEONEPASS
**Do the following things in parallel for** $i = 0, 1, 2, \ldots, \lfloor \log(|V|^3) \rfloor$ **:**
Let be $\widetilde{P}_i = 2^i$.
Uniformly choose one path of length 2 using algorithm UNIFORMSTAR($\widetilde{P}_i$).
**If** UNIFORMSTAR did not select a star until the end of the stream, **then return** $\perp$.
Let $(v_1, v_2, ..., v_\alpha)$ be this star with $v_1$ as middle node.
After choosing the path, test if each edge $(v_i, v_j)$ for $i, j \in \{2, \ldots, \alpha\}$ and $i \neq j$ appears within the rest of the stream.
**if** all edges appear in the stream after the incidence list of $v$ **then** set $\beta_i \leftarrow 1$
**else** set $\beta_i \leftarrow 0$
**in parallel do:** count the number $P = \sum_{v \in V} \binom{d_v}{\alpha - 1}$ of stars in the graph.
set $\beta \leftarrow \beta_{\lceil \log P \rceil}$
**return** $\beta$

Figure 9.15: The 1-pass algorithm SAMPLECLIQUEONEPASS for incidence streams

COUNTCLIQUESONEPASS ( $s \in \mathbb{N}$ )
Run $s$ instances of SAMPLECLIQUEONEPASS in parallel.
Let $s'$ be the number of instances not returning $\perp$.
Let $\beta_i$ be the value returned by the $i$th such instance (not returning $\perp$).
$\widetilde{K_\alpha} \leftarrow \left(\frac{1}{s'} \sum_{i=1}^{s'} \beta_i\right) \cdot |S_\alpha|/2 = \left(\frac{1}{s'} \sum_{i=1}^{s'} \beta_i\right) \cdot \sum_{v \in V} \binom{d_v}{\alpha - 1}/2$.
**return** $\widetilde{K_\alpha}$.

Figure 9.16: The 1-pass algorithm COUNTCLIQUESONEPASS for incidence streams

*If* $\epsilon \leq 1/2$ *and* $s \geq \frac{6}{\epsilon^2} \cdot \frac{|S_\alpha|}{|K_\alpha|} \cdot \ln(\frac{4}{\delta})$. *then with probability* $1 - \delta$ *the algorithm outputs a value* $\widetilde{K_\alpha}$ *satisfying*

$$(1 - \epsilon) \cdot |K_\alpha| \leq \widetilde{K_\alpha} \leq (1 + \epsilon) \cdot |K_\alpha| \ .$$

**Proof :** Equivalent to the proof of Lemma 9.2.5. □

Based on that we can again develop an algorithm COUNTCLIQUESONEPASSSAFE. It has a number $s$ and a desired error probability $\psi$ as input parameters and outputs a pair $(\widetilde{K_\alpha}, \tilde{\epsilon})$. The algorithm gives the guarantee that $\widetilde{K_\alpha}$ is a $(1 \pm \epsilon)$-approximation of $|K_\alpha|$ with probability $1 - \psi$ and uses at most $s$ parallel instances of SAMPLECLIQUEONEPASS. We show the pseudocode of COUNTCLIQUESONEPASSSAFE in Figure 9.17.

**Lemma 9.3.3** *Let* $(\widetilde{K_\alpha}, \tilde{\epsilon})$ *be the output of algorithm* COUNTCLIQUESONEPASSSAFE. *With probability* $1 - \psi$ *the following statements are true:*

COUNTCLIQUESONEPASSSAFE ( $s \in \mathbb{N}, \psi \in (0,1)$ )
Set $r \leftarrow \lceil 2 \cdot \log(4/\psi) \rceil$.
Set $t \leftarrow \lceil s/(4\log(4/\psi)) \rceil$.
Run $r$ instances of COUNTCLIQUESONEPASS($t$) in parallel.
Let $\widetilde{K_\alpha}^{(i)}$ be the value returned by the $i$th instance.
$\widetilde{K_\alpha} \leftarrow \text{median}_i(\widetilde{K_\alpha}^{(i)})$.
Set $\tilde{\epsilon} \leftarrow \sqrt{\frac{1056}{\widetilde{K_\alpha}} \cdot \frac{|S_\alpha|}{s} \cdot \log\frac{4}{\psi}} = \sqrt{\frac{1056}{\widetilde{K_\alpha}} \cdot \frac{\sum_{v \in V}\binom{d_v}{\alpha-1}}{s} \cdot \log\frac{4}{\psi}}$.
**return** $(\widetilde{K_\alpha}, \tilde{\epsilon})$.

Figure 9.17: The 1-pass algorithm COUNTCLIQUESONEPASSSAFE for adjacency streams

- $(1-\tilde{\epsilon}) \cdot |K_\alpha| < \widetilde{K_\alpha} < (1+\tilde{\epsilon}) \cdot |K_\alpha|$
- *If* $s \geq \frac{2112}{\epsilon^2} \cdot \frac{|S_\alpha|}{|K_\alpha|} \cdot \log(4/\psi)$*, then the algorithm outputs* $(\widetilde{K_\alpha}, \tilde{\epsilon})$ *with* $\tilde{\epsilon} \leq \epsilon$.

**Proof :** Equivalent to the proof of Lemma 9.1.4. □

**Theorem 28** *There is a 1-Pass streaming algorithm to count the number of* $K_\alpha$ *in incidence streams up to a multiplicative error of* $1 \pm \epsilon$ *with probability at least* $1-\psi$*, which needs*

$$O\left(\frac{1}{\epsilon^2} \cdot \left(1 + \frac{|S_\alpha|}{|K_\alpha|}\right) \cdot \log(\frac{1}{\psi}) \cdot \log^2|V|\right)$$

*memory bits and has amortized expected update time*

$$O\left(1 + \left(\frac{1}{\epsilon^2} \cdot \left(1 + \frac{|S_\alpha|}{|K_\alpha|}\right) \cdot \log\left(\frac{1}{\psi}\right) \cdot \frac{\log^2|V| \cdot |V|}{|E|}\right)\right) \quad .$$

## 9.4 Counting $K_{3,3}$ in Incidence Streams

We propose a method to estimate the number of $K_{3,3}$, when the graph is directed and given as an incidence stream and the outdegree of each node is bounded by $\Delta$. The stream of edges is ordered by destination nodes (so we see for each destination node all source nodes after one another). Our assumption is justified because in large social graphs and the webgraph there are often only a small number of links going out of each node. The graphs are often stored on hard disk(s) and for each node all incoming edges are precomputed and stored with the graph.

We do not assume any ordering by source nodes. Let $K_{3,3}$ denote the set of $K_{3,3}$ minors and $K_{3,1}$ denote the set of $K_{3,1}$ minors as shown in Figure 9.18.

We will first show how we can choose a $K_{3,1}$ uniformly at random from the stream. This is done similarly to choosing the length-2-paths in the triangle algorithm for incidence lists. We

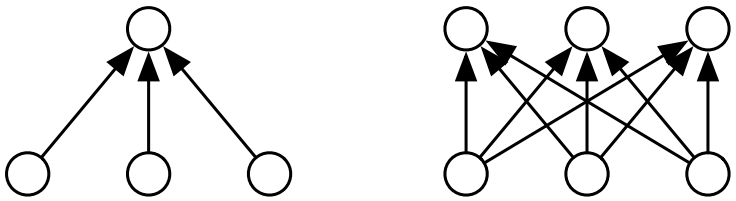

Figure 9.18: A $K_{3,1}$ (on the left) and a $K_{3,3}$ (on the right).

start a number of different estimations on the number of $K_{3,1}$. In parallel we count the number $|K_{3,1}| = \sum_{i=1}^{|V|} d_i \cdot (d_i - 1) \cdot (d_i - 2)/6$.

We will extend the method UNIFORMTWOPATH to a method UNIFORM$K_{3,1}$ as shown in Figure 9.19. It has an estimation $P$ of $|K_{3,1}|$ as input parameter and selects uniformly at random one $K_{3,1}$ motif from the incidence stream. Using UNIFORM$K_{3,1}$ we can develop a method SAMPLE$K_{3,3}$, outputting a variable $\beta$ whose expectation is related to the number $|K_{3,3}|$. The method SAMPLE$K_{3,3}$ is given in Figure 9.20. It can be implemented using $O(\log^2 |V|)$ memory bits.

UNIFORM$K_{3,1}(P)$
  Select value $k$ uniformly from the set $\{1, \ldots, P\}$.
  **For each** node $v$ in the incidence list **do**:
    **If** $k > 0$ **then**
      Set $h \leftarrow \lceil f^{-1}(k) \rceil$ (with $f(x) := \binom{x}{3}$)
      Set $k_2 \leftarrow k - f(h-1)$
      Set $i \leftarrow \left\lceil \sqrt{2k_2 + \frac{1}{4}} + \frac{1}{2} \right\rceil$
      Set $j \leftarrow i - \frac{i^2 - i}{2} + k_2 - 1$
      Pass over the complete incidence list of node $v$.
      **If** incidence list of $v$ contains more than $j$ edges **then**
        $a \leftarrow$ the $h$th node in the incidence list of $v$
        $b \leftarrow$ the $i$th node in the incidence list of $v$
        $c \leftarrow$ the $j$th node in the incidence list of $v$
        $u \leftarrow v$
      **end if**
      $d \leftarrow$ degree of node $v$
      $k \leftarrow k - \frac{d \cdot (d-1) \cdot (d-2)}{6}$
    **end if**
  **end do**
  **return** edges $(a, u)$,$(b, u)$ and $(c, u)$

Figure 9.19: The 1-pass algorithm UNIFORM$K_{3,1}$ for directed incidence streams

SAMPLE$K_{3,3}$
**Do the following things in parallel for** $i = 0, 1, 2, \ldots, \lfloor \log(|V|^4) \rfloor$ **:**
Let be $\widetilde{P}_i = 2^i$.
From all $K_{3,1}$ occuring in the stream choose one uniformly using UNIFORM$K_{3,1}(\widetilde{P}_i)$.
**If** UNIFORM$K_{3,1}$ did not select a $K_{3,1}$ until the end of the stream, **then return** $\perp$.
Let the three edges of the chosen $K_{3,1}$ be $(a, u)$, $(b, u)$ and $(c, u)$
Select uniformly $x_1, x_2 \in \{a, b, c\}$
Choose uniformly random variables $k_1, k_2 \in \{1, 2, \ldots \Delta\}$
**If** $k_1 = k_2 \wedge x_1 = x_2$ **then** set $\beta_i \leftarrow 0$
**else:**
Go on passing over the rest of stream (the part behind the occurence of the $K_{3,1}$).
Select $(x_1, v)$ as the $k_1$-th edge $(x_1, \cdot)$ after selecting the $K_{3,1}$.
Select $(x_2, w)$ as the $k_2$-th edge $(x_2, \cdot)$ after selecting the $K_{3,1}$.
From the time of selecting $(x_1, v)$:
check, if $(a, v)$,$(b, v)$,$(c, v)$ are present in the stream
From the time of selecting $(x_2, w)$:
check, if $(a, w)$,$(b, w)$,$(c, w)$ are present in the stream
**If** both is the case, **then** set $\beta_i \leftarrow 1$ **else** set $\beta_i \leftarrow 0$.
**In parallel to the for loop** count the number $|K_{3,1}| = P = \sum_{v \in V} \binom{d_v}{3}$ of $K_{3,1}$ in the graph.
set $\beta \leftarrow \beta_{\lceil \log P \rceil}$
**return** $\beta$

Figure 9.20: The 1-pass algorithm SAMPLE$K_{3,3}$ for directed incidence streams

**Lemma 9.4.1** *Algorithm* SAMPLE$K_{3,3}$ *outputs with probability at least* $1/2$ *a random value* $\beta$ *having*

$$\mathbf{E}[\beta] = \frac{2 \cdot |K_{3,3}|}{9 \cdot \Delta^2 \cdot |K_{3,1}|} \ .$$

*Otherwise it outputs the value* $\perp$.

**Proof :** We set $\beta = \beta_i$ with $i = \lceil \log P \rceil$ at the end of the algorithm. This value of $\beta_i$ has been set to 0 or 1, if UNIFORM$K_{3,1}(\widetilde{P}_i)$ did select a $K_{3,1}$. Since $\widetilde{P}_i = 2^{\lceil \log P \rceil}$ we have $P \leq \widetilde{P}_i < 2 \cdot P$.

UNIFORM$K_{3,1}(\widetilde{P}_i)$ selects a $K_{3,1}$ by first choosing $k \in \{1, \ldots, \widetilde{P}_i\}$ uniformly at random and then selecting the k-th $K_{3,1}$ in the stream. If $k \leq \widetilde{P}_i/2$ we have $k \leq P$ and therefore a $K_{3,1}$ is selected. This happens with probability $1/2$ and in that case SAMPLE$K_{3,3}$ does not output $\perp$.

Let us now condition on the event that $\beta \neq \perp$ and analyse the expected value of $\beta$. Let $(a, b, c, u, v, w)$ be an arbitrary fixed $K_{3,3}$ with edges directed from a,b,c to u,v and w. Let $u$ be the vertex whose incidence list appears first within the incidence stream, $v$, $w$ occuring after $u$ withing the stream. The $K_{3,3}$ will be detected exactly when all of the following events occur:

- $a, b, c, u$ are chosen as $K_{3,1}$ with $u$ being the destination node
- $v$ and $w$ must be chosen
- $x_1$ must be the first within the incidence list of $v$.
- $x_2$ must be the first within the incidence list of $w$.

The probability of the first event is $1/|K_{3,1}|$.
Conditioned on the first event the probability to choose $v$ and $w$ is $2/\Delta^2$: Each edge $(x_1, \cdot)$ appearing after $(x_1, u)$ in the stream has a probability of $1/\Delta$ to be chosen by the algorithm. We know that $(x_1, v)$ and $(x_1, w)$ appear after $(x_1, u)$ in the stream. Therefore each of these two edges has a probability of $1/\Delta$ to be chosen. By a similar argument we have independently a probability of $1/\Delta$ to choose the edge $(x_2, v)$ resp. $(x_2, w)$. We select the nodes $v$ and $w$ if we either select $(x_1, v)$ and $(x_2, w)$ or $(x_1, w)$ and $(x_2, v)$. Therefore the probability to choose $v$ and $w$ is $2/\Delta^2$.

Observe that the probability for $v$ and $w$ to be chosen does not depend on the choice of $x_1$ and $x_2$. We can therefore exchange the order in which we analyse these events.
Conditioned on the first two events the probability for the third event is $1/3$, also the probability for the fourth event. We get alltogether a probability of $\frac{2}{9 \cdot \Delta^2 \cdot |K_{3,1}|}$ to choose the fixed $K_{3,3}$. □

A streaming algorithm COUNT$K_{3,3}$, which outputs an estimate of $|K_{3,3}|$, easily follows. It can be adjusted using an input parameter $s$. It is given in Figure 9.21.

COUNT$K_{3,3}$ ( $s \in \mathbb{N}$ )
Run $s$ instances of SAMPLE$K_{3,3}$ in parallel.
Let $s'$ be the number of instances not returning $\perp$.
Let $\beta_i$ be the value returned by the $i$th such instance (not returning $\perp$).
$\widetilde{K_{3,3}} \leftarrow \left(\frac{1}{s'} \sum_{i=1}^{s'} \beta_i\right) \cdot \frac{9}{2} \cdot \Delta^2 \cdot |K_{3,1}| = \left(\frac{1}{s'} \sum_{i=1}^{s'} \beta_i\right) \cdot \frac{9}{2} \cdot \Delta^2 \cdot \sum_{v \in V} \binom{d_v}{3}$.
**return** $\widetilde{K_{3,3}}$.

Figure 9.21: The 3-pass algorithm COUNT$K_{3,3}$ for adjacency streams

**Lemma 9.4.2** *Algorithm* COUNT$K_{3,3}$ *outputs a value* $\widetilde{K_{3,3}}$ *having expected value* $\mathbf{E}[\widetilde{K_{3,3}}] = |K_{3,3}|$. *If* $\epsilon \leq 1/2$ *and* $s \geq \frac{54}{\epsilon^2} \cdot \frac{\Delta^2 \cdot |K_{3,1}|}{|K_{3,3}|} \cdot \ln(\frac{2}{\delta})$ *then with probability* $1 - \delta$ *the algorithm outputs a value* $\widetilde{K_{3,3}}$ *satisfying*

$$(1 - \epsilon) \cdot |K_{3,3}| \leq \widetilde{K_{3,3}} \leq (1 + \epsilon) \cdot |K_{3,3}| \ .$$

**Proof :** Equivalent to the proof of Lemma 9.2.5. □

COUNT$K_{3,3}$SAFE ( $s \in \mathbb{N}, \psi \in (0,1)$ )
Set $r \leftarrow \lceil 2 \cdot \log(4/\psi) \rceil$.
Set $t \leftarrow \lceil s/(4\log(4/\psi)) \rceil$.
Run $r$ instances of COUNT$K_{3,3}$ $(t)$ in parallel.
Let $\widetilde{K_{3,3}}^{(i)}$ be the value returned by the $i$th instance.
Set $\widetilde{K_{3,3}} \leftarrow \text{median}_i(\widetilde{K_{3,3}}^{(i)})$.
Set $\tilde{\epsilon} \leftarrow \sqrt{\frac{9504 \cdot \Delta^2}{\widetilde{K_{3,3}}} \cdot \frac{|K_{3,1}|}{s} \cdot \log \frac{4}{\psi}} = \sqrt{\frac{9504 \cdot \Delta^2}{\widetilde{K_{3,3}}} \cdot \frac{\sum_{v \in V} \binom{d_v}{3}}{s} \cdot \log \frac{4}{\psi}}$.
**return** $(\widetilde{K_{3,3}}, \tilde{\epsilon})$.

Figure 9.22: The 3-pass algorithm COUNT$K_{3,3}$SAFE for adjacency streams

We can develop an algorithm COUNT$K_{3,3}$SAFE based on COUNT$K_{3,3}$. It has a number $s$ and a desired error probability $\psi$ as input parameters and outputs a pair $(\widetilde{K_{3,3}}, \tilde{\epsilon})$. The algorithm gives the guarantee that $\widetilde{K_{3,3}}$ is a $(1 \pm \epsilon)$-approximation of $|K_{3,3}|$ with probability $1 - \psi$. It uses at most $s$ parallel instances of SAMPLE$K_{3,3}$. We show the pseudocode of COUNT$K_{3,3}$SAFE in Figure 9.22.

**Lemma 9.4.3** *Let $(\widetilde{K_{3,3}}, \tilde{\epsilon})$ be the output of algorithm* COUNT$K_{3,3}$SAFE. *With probability $1 - \psi$ the following statements are true:*

- $(1 - \tilde{\epsilon}) \cdot |K_{3,3}| < \widetilde{K_{3,3}} < (1 + \tilde{\epsilon}) \cdot |K_{3,3}|$
- *If $s \geq \frac{19008 \cdot \Delta^2}{\epsilon^2} \cdot \frac{|K_{3,1}|}{|K_{3,3}|} \cdot \log(4/\psi)$, then the algorithm outputs $(\widetilde{K_{3,3}}, \tilde{\epsilon})$ with $\tilde{\epsilon} \leq \epsilon$.*

**Proof :** Equivalent to the proof of Lemma 9.1.4. □

**Theorem 29** *There is a 1-Pass streaming algorithm to count the number of $K_{3,3}$ in incidence streams ordered by destination nodes with outdegree bounded by $\Delta$ up to a multiplicative error of $\epsilon$ with probability at least $1 - \psi$, which needs*

$$O\left(\log^2(|V|) \cdot \frac{|K_{3,1}| \cdot \Delta^2 \ln(\frac{1}{\psi})}{|K_{3,3}| \cdot \epsilon^2}\right)$$

*memory bits.*

# 10 Conclusions

In this thesis we developed new methods to analyse dynamic geometric data streams and obtain structural information about the large data sets encoded in the streams.

In Chapter 3 we first developed a method to draw a uniform random sample from a multiset, when the multiset is given as a turnstyle data stream. The method we propose is a building block for various data stream algorithms. As examples we showed in Chapter 4 some direct consequences of the new sampling method, i.e. how to maintain $\epsilon$-nets and $\epsilon$-approximations of points when the point set is given as a dynamic geometric data stream. We also developed a method to estimate the weight of a minimum tree spanning all the points encoded in a dynamic geometric data stream. As random sampling and $\epsilon$-approximations are powerful tools in computational geometry we believe that our techniques have many more applications.

The space used by the algorithm for $\epsilon$-approximations, i.e. roughly $O(1/\epsilon^2)$, is essentially optimal as a function of $\epsilon$. This is, because it is known that, for some range spaces, the size of $\epsilon$-approximations tends to $1/\epsilon^2$ as the VC dimension tends to infinity. However, for some range spaces smaller $\epsilon$-approximations can be constructed, even for points delivered in an insertions-only stream. For example, [118] showed how to compute $\epsilon$-approximations for ranges defined by halfspaces in $d$ dimensions of size roughly $O(1/\epsilon^{2-2/(d+1)})$. We do not know how to extend this result to dynamic geometric data streams.

To estimate the weight of the minimum spanning tree of the points encoded in a dynamic geometric data stream we used $O(\log^3(1/\delta) \cdot (\log(\Delta)/\epsilon)^{O(d)})$ space. Although we give the first algorithm at all to estimate the value in space polylogarithmic in $\Delta$, we believe that one can develop algorithms having much better memory bounds.

One of the central results of this thesis, a universal method to construct coresets for k-median, k-means, MaxCut, and more problems, has been given in Chapter 5. Our method is much simpler than previous coreset methods [8, 60, 61] and makes less assumptions about the distribution of the points. The only information needed to construct coresets is the number of points in heavy cells (cells containing a certain number of points) of certain square grids. The simplicity of our method enabled us to develop the fastest known PTAS for Euclidean MaxCut in Section 5.5.3 and the first efficient methods to maintain coresets for k-median, k-means, MaxCut, and more problems on dynamic geometric data streams in Chapter 6.

The coreset we obtain is of size $O(k \cdot \log n/\epsilon^{d+1})$ for k-median, $O(k \cdot \log n/\epsilon^{d+2})$ for k-means and $O(\log n/\epsilon^{d+1})$ for all other problems we consider. Recently some methods have been proposed to obtain smaller coresets for k-median and k-means. Har-Peled and Kushal [60] showed how to compute $\epsilon$-coresets of size $O(k^2/\epsilon^d)$ for k-median and of size $O(k^3/\epsilon^{d+1})$ for k-means (not dependent on $n$). However, their coreset construction does not apply to dynamic geometric data streams. Their results indicate that the space dependency of our algorithms on $n$

could also be improved.

All space requirements of our coreset algorithms depend exponentially on $d$ (we did not state it in the formulas because we assumed a constant dimension). Therefore our methods are not suited for high dimensional data. Recently Chen [26] proposed a method to obtain coresets using space which depends polynomially on $d$. His method is suited for streams of insertions of points, but does not translate to streams of insertions and deletions. It remains an interesting open problem to develop coreset methods for dynamic geometric data streams, which have space complexity polynomial in the dimension $d$.

In Chapter 7 we used our coreset technique to develop the first kinetic data structure for the Euclidean MaxCut problem. Our KDS can be extended to MaxTSP, MaxMatching, and average distance. However, the time to compute a solution from the coreset (which has to be done for each query to the data structure, or, alternatively with each event) can differ significantly.

Extending our KDS to $k$-median and $k$-means clustering requires additional ideas. The technical problem is here that one cannot get a lower bound on the solution from the width of the bounding box. Hence, it is not clear how to get an upper bound on the number of events. Developing kinetic data structures for $k$-median and $k$-means therefore remains an interesting open problem.

In Chapter 8 we presented an efficient implementation of a $k$-means clustering algorithm using coresets. Our algorithm performs very well compared to KMHybrid [105] for small dimension and small to medium $k$. The quality of the solutions varies less than that of KMHybrid, which implies that we need fewer runs to guarantee a good solution. The main strength of our algorithm is to quickly find relatively good approximations for many values of $k$, for example when a good value for $k$ is not known in advance. In this case, we can also use the coresets to compute the average clustering coefficient and thus to find a good choice of $k$.

As mentioned above recently some constructions for smaller coresets [26, 60] have been developed. It would be interesting to measure if the usage of these smaller coresets would lead to even faster convergence of $k$-means based algorithms.

We have proposed a methodology in Chapter 9 to find $(1\pm\epsilon)$-approximations on the number of frequent subgraphs of large graphs given as data streams. The amount of samples resp. memory bits needed by our algorithms depend on the number of certain small structures in these graphs. Recent results on the internal structure of the webgraph or large sozial graphs [17, 81, 89, 92, 78] suggest that the amount of space needed by our algorithm to count motifs is constant or at most logarithmic in the number of nodes for these graphs. Recent tests [17] suggest that our algorithms can compute good estimations on the number of triangles of real webgraph crawls in time comparable to the time to read the graph from the hard disc.

# Bibliography


[1] P. K. Agarwal, J. Gao, and L. J. Guibas. Kinetic Medians and kd-Trees. *Proceedings of the 10th Annual European Symposium on Algorithms (ESA)*, pp. 5–16, 2002.

[2] P. K. Agarwal, S. Har-Peled, and K. R. Varadarajan. Approximating Extent Measures of Points. *Journal of the ACM*, 51(4):606–635, July 2004.

[3] P. Agarwal, S. Har-Peled, and K. Varadarajan. Geometric Approximation via Coresets. Survey available at http://valis.cs.uiuc.edu/ sariel/research/papers/04/survey/survey.pdf

[4] N. Alon, Y. Matias, and M. Szegedy. The Space Complexity of Approximating the Frequency Moments. *J. Comput. Syst. Sci.*, 58(1), pp. 137–147, 1999.

[5] K. Alsabti, S. Ranka, and V. Singh. An Efficient k-Means Clustering Algorithm. *Proceedings of the first Workshop on High Performance Data Mining*, 1998.

[6] D. Arthur and S.Vassilvitskii. How Slow is the k-Means Method? *Proceedings of the 21nd Annual ACM Symposium on Computational Geometry (SoCG)*, pp.144–153, 2006.

[7] M. Badoiu and K. Clarkson. Smaller Core-Sets for Balls. *Proceedings of the 14th Symposium on Discrete Algorithms* (SODA'03), pp. 801–802, 2003.

[8] M. Badoiu, S. Har-Peled, and P. Indyk. Approximate Clustering via Coresets. *Proceedings of the 34th Annual ACM Symposium on Theory of Computing* (STOC'02), pp. 250–257, 2002.

[9] A. Bagchi, A. Chaudhary, D. Eppstein and M. T. Goodrich. Deterministic Sampling and Range Counting in Geometric Data Streams. *Proceedings of the 20th Annual Symposium on Computational Geometry (SoCG)*, pp. 144–151, 2004.

[10] Z. Bar-Yossef, T.S. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan. Counting Distinct Elements in a Data Stream. Proceedings of the 6th International Workshop on Randomization and Approximation Techniques, pages 1-10, 2002.

[11] J. Basch. *Kinetic Data Structures*. Ph.D. thesis, Stanford University, 1999.

[12] J. Basch, L. J. Guibas, and J. Hershberger. Data Structures for Mobile Data. *J. Algorithms*, 31(1):1–28 1999.

[13] J. Basch, L. J. Guibas, and G. Ramkumar. Sweeping Lines and Line Segments with a Heap. *Proceedings of the 13th Annual ACM Symposium on Computational Geometry (SoCG)*, pp. 469–471, 1997.

[14] P. Berkhin. Survey of Clustering Data Mining Techniques. Available at ..., 2002.

[15] S. Bespamyatnikh, B. Bhattacharya, D. Kirkpatrick, and M. Segal. Mobile Facility Location. *Proceedings of the 4th DIAL M*, pp. 46–53, 2000.

[16] G. S. Brodal and R. Jacob. Dynamic Planar Convex Hull. *Proceedings of the 43rd IEEE Symposium on Foundations of Computer Science (FOCS)*, pp. 617–626, 2002.

[17] L. Buriol, G. Frahling, S. Leonardi, A. Marchetti-Spaccamela, and C. Sohler. Counting Triangles in Data Streams. *Proceedings of the 25th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 253–262, 2006

[18] J. Carter and M. Wegman. Universal Classes of Hash Functions. *Journal of Computer and System Sciences*, 18(2), pp. 143–154, 1979

[19] T. M. Chan. Faster Coreset Constructions and Data Stream Algorithms in Fixed Dimensions. *Proceedings of the 20th Annual Symposium on Computational Geometry (SoCG)*, pp. 152–159, 2004.

[20] T. Chan, B. Sadjad. Geometric Optimization Problems Over Sliding Windows. *Proceedings of the 15th Annual International Symposium on Algorithms and Computation (ISAAC)*, pp. 246–258, 2004.

[21] M. Charikar, L. O'Callaghan, and R. Panigrahy. Better Streaming Algorithms for Clustering Problems. *Proceedings of the 35th Annual ACM Symposium on Theory of Computing (STOC)*, pp. 30–39, 2003.

[22] M. Charikar, S. Chaudhuri, R. Motwani, and V. Narasayya. Towards Estimation Error Guarantees for Distinct Values. *Proceedings of the 19th ACM SIGMOD Symposium on Principles of Database Systems (PODS)*, 2000.

[23] M. Charikar, C. Chekuri, T. Feder, and R. Motwani. Incremental Clustering and Dynamic Information Retrieval. *Proceedings of the 29th Annual ACM Symposium on Theory of Computing (STOC)*, 626–635, 1997.

[24] M. Charikar, K. Chen, M. Farach-Colton. Finding Frequent Items in Data Streams. *Proceedings of the . 29th Annual International Colloquium on Automata, Languages and Programming (ICALP)*, pp. 693–703, 2002.

[25] B. Chazelle, R. Rubinfeld, and L. Trevisan. Approximating the Minimum Spanning Tree Weight in Sublinear Time. *Proceedings of the 28th Annual International Colloquium on Automata, Languages and Programming (ICALP)*, pages 190–200, 2001.

[26] K. Chen. On k-Median Clustering in High Dimensions. *Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1177–1185, 2006.

[27] D. Coppersmith and S. Winograd. Matrix Multiplication via Arithmetic Progressions. *Journal of Symbolic Computation* 3, no. 9.

[28] G. Cormode, M. Datar, P. Indyk. Comparing Data Streams using Hamming norms. *Proceedings of the International Conference on Very Large Databases (VLDB)*, pp. 335–345, 2002.

[29] G. Cormode and S. Muthukrishnan. Radial Histograms for Spatial Streams. DIMACS Technical Report 2003-11, 2003.

[30] G. Cormode and S. Muthukrishnan. Improved Data Stream Summaries: The Count-Min Sketch and its Applications. *Proceedings of the 6th Latin American Theoretical Informatics (LATIN)*, pp. 29–38, 2004.

[31] G. Cormode and S. Muthukrishnan and I. Rozenbaum. Summarizing and Mining Inverse Distributions on Data Streams via Dynamic Sampling. *DIMACS Technical Report 2005-11*, 2005.

[32] A. Czumaj, F. Ergün, L. Fortnow, A. Magen, I. Newman, R. Rubinfeld, and C. Sohler. Sublinear-Time Approximation of Euclidean Minimum Spanning Tree. *SIAM Journal on Computing*, 35(1): 91-109 ,2005.

[33] A. Czumaj and C. Sohler. Estimating the Weight of Metric Minimum Spanning Trees in Sublinear-Time. *Proceedings of the 36th Annual ACM Symposium on Theory of Computing (STOC)*, pp. 175–183, 2004.

[34] A. Czumaj and C. Sohler. Sublinear-Time Approximation for Clustering via Random Sampling. *Proceedings of the 31st Annual International Colloquium on Autonate, Languages and Programming* (ICALP'04), pp. 396–407, 2004.

[35] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. D. Ullman. Computing Iceberg Queries Efficiently. *Proceedings of the 1998 Intl. Conf. on Very Large Data Bases*, pp. 299-310, 1998.

[36] J. Feigenbaum, S. Kannan, and J. Zhang. Computing Diameter in the Streaming and Sliding Window Models. Technical Report YALEU/DCS/TR-1245, Yale University, 2002.

[37] W. Fernandez de la Vega, M. Karpinski, C. Kenyon. Approximation Schemes for Metric Minimum Bisection and Partitioning. *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2004.

[38] W. Fernandez de la Vega, M. Karpinski, C. Kenyon, and Y. Rabani. Approximation Schemes for Clustering Problems. *Proceedings of the 35th Annual ACM Symposium on Theory of Computing (STOC)*, pp. 50–58, 2003.

[39] W. Fernandez de la Vega and C. Kenyon. A Randomized Approximation Scheme for Metric MAX-CUT. J. Comput. Syst. Sci., 63(4):531-541, 2001.

[40] P. Flajolet and G. Martin. Probabilistic Counting Algorithms for Data Base Applications. Journal of Computer and System Sciences, 31:182–209, 1985.

[41] E. Forgey. Cluster Analysis of Multivariate Data: Efficiency vs. Interpretability of Classification. *Biometrics*, 21:768, 1965.

[42] G. Frahling, P. Indyk, and C. Sohler. Sampling in Dynamic Data Streams and Applications. *Proceedings of the 21st Annual Symposium on Computational Geometry (SoCG)*, pages 142–149, 2005. Invited to the special issue of SoCG 2005, to appear in *International Journal of Computational Geometry and Applications (IJCGA)*.

[43] G. Frahling and C. Sohler. Coresets in Dynamic Geometric Data Streams. *Proceedings of the 37th Annual ACM Symposium on Theory of Computing (STOC)*, pp. 209–217, 2005.

[44] G. Frahling and C. Sohler. A Fast k-Means Implementation using Coresets. *Proceedings of the 22nd Annual Symposium on Computational Geometry (SoCG)*, pages 135–143, 2006. Invited to the special issue of SoCG 2006, to appear in *International Journal of Computational Geometry and Applications (IJCGA)*.

[45] H.N. Gabow. Data Structures for Weighted Matching and Nearest Common Ancestors with Linking *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 434–443, 1990.

[46] S. Ganguly, M. Garofalakis and R. Rastogi. Tracking Set-Expression Cardinalities over Continuous Update Streams. The VLDB Journal, 13(4), pp. 354–369, 2004.

[47] J. Gao, L. J. Guibas, J. Hershberger, L. Zhang, and A. Zhu. Discrete Mobile Centers. *Discrete & Computational Geometry*, 30(1):45–63, 2003.

[48] A. Gilbert, S. Guha, Y. Kotidis, P. Indyk, S. Muthukrishnan, M. Strauss. Fast, Small Space algorithm for Approximate Histogram Maintenance. *Proceedings of the 34th Annual ACM Symposium on Theory of Computing (STOC)*, pp.389–398, 2005.

[49] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss, Surfing Wavelets on Streams: One-Pass Summaries for Approximate Aggregate Queries. *VLDB*, 2001, pp. 79–88.

[50] M. X. Goemans and D. P. Williamson. Improved Approximation Algorithms for Maximum Cut and Satisfiability Problems using Semidefinite Programming *JACM*, 42:1115–1145, 1995.

[51] O. Goldreich, S. Goldwasser, D. Ron. Property Testing and its Connection to Learning and Approximation. *Journal of the ACM*, 45(4):653–750, 1998.

[52] J. Goodman and J. O'Rourke. Handbook of Discrete and Computational Geometry. CRC Press, 1997.

[53] S. Guha, N. Koudas, and K. Shim. Data-Streams and Histograms. *Proceedings of the Annual ACM Symposium on Theory of Computing (STOC)*, 2001, pp. 471–475.

[54] S. Guha, N. Mishra, R. Motwani, and L. O'Callaghan. Clustering Data Streams. *Proceedings of the 41st IEEE Symposium on Foundations of Computer Science (FOCS)*, 359–366, 2000.

[55] L. J. Guibas. Kinetic Data Structures — a State of the Art Report. *Proceedings of the 3rd Workshop on the Algorithmic Foundations of Robotics (WAFR)*, pp. 191–209, 1998.

[56] L. J. Guibas. Modeling Motion. In *Handbook of Discrete and Computational Geometry*, edited by J. E. Goodman and J. O'Rourke, 2nd edition, Chapter 50, pp. 1117–1134, 2004.

[57] P. Haas, J. Naughton, S. Seshadri, and L. Stokes. Sampling-based Estimation of the Number of Distinct Values of an Attribute. *Proceedings of the 21st International Conference on Very Large Data Bases (VLDB)*, pp.311–322, 1995.

[58] T. Hagerub and C. Rüb. A Guided Tour of Chernoff Bounds. Information Processing Letters, 33:305–308, 1989/90.

[59] S. Har-Peled. Clustering Motion. *Discrete & Computational Geometry*, 31:545–565, 2004.

[60] S. Har-Peled and A. Kushal. Smaller Coresets for k-Median and k-Means Clustering.

[61] S. Har-Peled and S. Mazumdar. Coresets for k-Means and k-Medians and Their Applications. *Proceedings of the 36th Annual ACM Symposium on Theory of Computing (STOC)*, 291–300, 2004.

[62] S. Har-Peled and B. Sadri. On Lloyd's k-Means Method. *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2005.

[63] S. Har-Peled and K. Varadarajan. Projective Clustering in High Dimensions using Coresets. *Proceedings 18th Annual ACM Symposium on Computational Geometry* (SoCG'02), pp. 312–318, 2002.

[64] F. Harary and H. J. Kommel. Matrix Measures for Transitivity and Balance. *Journal of Mathematical Sociology* (6), 199210.

[65] J. Hartigan. Clustering Algorithms. *John Wiley & Sons*, New York, 1975.

[66] D. Haussler, E: Welzl. ϵ-Nets and Simplex Range Queries. *Discrete and Computational Geometry*, 2:127–151, 1987.

[67] M. Henzinger, P. Raghavan, and S. Rajagopalan. Computing on Data Streams. 1998

[68] J. Hershberger. Smooth Kinetic Maintenance of Clusters. *Computational Geometry, Theory and Applications*, 31(1–2):3–30, 2005.

[69] J. Hershberger and S. Suri. Convex Hulls and Related Problems in Data Streams. Proceedings of the ACM/DIMACS Workshop on Management and Processing of Data Streams, 2003.

[70] J. Hershberger and S. Suri. Adaptive Sampling for Geometric Problems over Data Streams. *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, 2004.

[71] M. Inaba, N. Katoh, and H. Imai. Applications of Weighted Voronoi Diagrams and Randomization to Variance-Based k-Clustering. *Proceedings of the 10th Annual ACM Symposium on Computational Geometry (SoCG)*, pp. 332–339, 1994.

[72] P. Indyk. High-Dimensional Computational Geometry. *Ph.D. thesis*, Stanford University, 2000.

[73] P. Indyk. Stable Distributions, Pseudorandom Generators, Embeddings and Data Stream Computation. *Proceedings of the 41st IEEE Symposium on Foundations of Computer Science (FOCS)*, pp. 189–197, 2000.

[74] P. Indyk. Better Algorithms for High-Dimensional Proximity Problems via Asymmetric Embeddings. Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 539–545, 2003.

[75] P. Indyk. Algorithms for Dynamic Geometric Problems over Data Streams. *Proceedings of the 36th Annual ACM Symposium on Theory of Computing (STOC)*, pp. 373–380, 2004.

[76] P. Indyk and D. Woodruff. Tight Lower Bounds for the Distinct Elements Problem. *Annual Symposium on Foundations of Computer Science*, pages 283–290, 2003.

[77] P. Indyk and D. Woodruff. Optimal Approximations of the Frequency Moments of Data Streams. *Proceedings of the 37th Annual ACM Symposium on Theory of Computing (STOC)*, 2005.

[78] S. Itzkovitz, N. Kashtan, D. Chklovskii, R. Milo, S. Shen-Orr, and U. Alon. Network Motifs: Simple Building Blocks of Complex Networks. *Science* (298), no. 509, 824 – 827.

[79] H. Jagadish, N. Koudas, and S. Muthukrishnan. Mining deviants in a time serias database. *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pp. 102–113, 1999.

[80] A. Jain and R. Dubes. Algorithms for Clustering Data. *Prentice Hall*, New Jersey, 1988.

[81] Hossein Jowhari and Mohammad Ghodsi. New Streaming Algorithms for Counting Triangles in Graphs *Proceedings of the COCOON*, 2005, pp. 710–716.

[82] T. Kanungo, D. Mount, N. Netanyahu, C. Piatko, R. Silverman, and A. Wu. An Efficient k-Means Clustering Algorithm: Analysis and Implementation. IEEE Trans. Pattern Anal. Mach. Intell. 24(7): 881-892, 2002.

[83] T. Kanungo, D. Mount, N. Netanyahu, C. Piatko, R. Silverman, and A. Wu. A Local Search Approximation Algorithm for k-Means Clustering. *Proceedings of the 18th Annual Symposium on Computational Geometry* (SoCG'02), pp. 10–18, 2002.

[84] H. Kaplan, R. E. Tarjan, and K. Tsioutsiouliklis. Faster Kinetic Heaps and Their Use in Broadcast scheduling. SODA, pp. 834–844, 2001.

[85] S. Khot, G. Kindler, E. Mossel, and R. O'Donnell. Optimal Inapproximability Results for Max-Cut and Other 2-Variable CSPs? *Proceedings of the 45th IEEE Symposium on Foundations of Computer Science (FOCS)*, pp. 146–154, 2004.

[86] D. Knuth. The Art of Computer Programming: Sorting and Searching, Vol. 3, Addison-Wesley, 1973.

[87] S. G. Kolliopoulos and S. Rao. A Nearly Linear-Time Approximation Scheme for the Euclidean k-Median Problem. *Proceedings of the 7th Annual European Symposium on Algorithms (ESA)*, pp. 378-389, 1999.

[88] R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. Trawling the Web for Emerging Cyber Communities. (1999), 403–416.

[89] R. Kumar, P. Raghavan, S. Rajagopalan, D. Sivakumar, A. Tomkins, and E. Upfal. Random Graph Models for the Web Graph. *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, 2000, pp. 57–65.

[90] A. Kumar, Y. Sabharwal, and S. Sen. A Simple Linear Time $(1+\varepsilon)$-Approximation Algorithm for k-Means Clustering in any Dimensions. *Proceedings of the 45th IEEE Symposium on Foundations of Computer Science (FOCS)*, pp. 454–462, 2004.

[91] A. Kumar, Y. Sabharwal, and S. Sen. Linear Time Algorithms for Clustering Problems in any Dimensions. *Proceedings of the 32nd Annual International Colloquium on Automata, Languages and Programming (ICALP)*, pp. 1374–1385, 2005.

[92] L. Laura, S. Leonardi, S. Millozzi, and J.F. Sybeyn. Algorithms and Experiments for the Webgraph. *Proceedings of the Annual European Symposium on Algorithms (ESA)*, 2002.

[93] S. Leonardi S. Millozzi L.S. Buriol, D. Donato. Link and Temporal Analysis of Wikigraphs. Technical Report (2005).

[94] Y. Linde, A. Buzo, and R. Gray. An algorithm for Vector Quantizer Design. *IEEE Transaction on Communications*, 28(1), pp. 84–94, 1980.

[95] S. Lloyd. Least Squares Quantization in PCM. *IEEE Transactions on Information Theory*, 28: 129–137, 1982.

[96] P. Lyman and H. Varian. How much information. *University of California, Berkeley, http://www2.sims.berkeley.edu/research/projects/how-much-info-2003/*, 2003.

[97] J. MacQueen. Some Methods for Classification and Analysis of Multivariate Observations. *Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pp. 281–296, 1967.

[98] G. S. Manku, R. Motwani. Approximate Frequency Counts over Data Streams. *Proceedings of the 2002 Intl. Conf. on Very Large Data Bases*, pp. 346–357, 2002.

[99] J. Matoušek. On Approximate Geometric k-Clustering. *Discrete & Computational Geometry*, 24(1): 61–84, 2000.

[100] R. Mettu and G. Plaxton. Optimal Time Bounds for Approximate Clustering. *Machine Learning*, 56(1-3):35–60, 2004.

[101] A. Meyerson. Online Facility Location. *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, pp. 426–431, 2001.

[102] N. Mishra, D. Oblinger, and L. Pitt. Sublinear Time Approximate Clustering. *Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 439–447, 2001.

[103] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[104] D. Mount. KMlocal: A Testbed for k-means Clustering Algorithms. Available at http://www.cs.umd.edu/ mount/Projects/KMeans/km-local-doc.pdf

[105] S. Muthukrishnan. Data streams: Algorithms and applications. *Foundations and Trends in Theoretical Computer Science*, Volume 1, Issue 2, 2005.

[106] N. Nisan. Pseudorandom Generators for Space-Bounded Computation. *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing (STOC)*, 204–212, 1990.

[107] A. Ostlin and R. Pagh. Uniform Hashing in Constant Time and Linear Space. *Proceedings of the 35th Annual ACM Symposium on Theory of Computing (STOC)*, ACM Press, 2003, pp. 622–628.

[108] D. Pelleg and A. Moore. Accelerating Exact k-Means Algorithms with Geometric Reasoning. *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 277–281, 1999.

[109] D. Pelleg and A. Moore. x-Means: Extending k-Means with Efficient Estimation of the Number of Clusters. *Proceedings of the 17th International Conference on Machine Learning*, 2000.

[110] S. Phillips. Acceleration of k-Means and Related Clustering Problems. *Proceedings of Algorithms Engineering and Experiments* (ALENEX'02), 2002.

[111] R. Prim. Shortest Connection Networks and some Generalizations. *Bell Systems Technical Journal*, 36:1389-1401, 1957.

[112] T. Schank and D. Wagner. Finding, Counting, and Listing all Triangles in Large Graphs, an Experimental Study. *Proceedings of the WEA*, 2005, pp. 606–609.

[113] J. Schmidt, A. Siegel, and A. Srinivasan. Chernoff-Hoeffding Bounds for Applications with Limited Independence. *SIAM Journal on Discrete Mathematics*, 8(2):223–250, 1995.

[114] R. Seidel and C. Aragon. Randomized Search Trees. Algorithmica 16, 464–497, 1996.

[115] S. Selim and M. Ismail. k-Means-Type Algorithms: A Generalized Convergence Theorem and Characterizations of Local Optimality. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6::81–87, 1984.

[116] D. Shasha, J. Tsong-Li Wang, and R. Giugno. Algorithmics and Applications of Tree and Graph Searching. *Proceedings of the 21st ACM SIGMOD Symposium on Principles of Database Systems (PODS)*, 2002,pp. 39–52.

[117] D. Sivakumar, Z. Bar-Yosseff, R. Kumar. Reductions in Streaming Algorithms, with an Application to Counting Triangles in Graphs. *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)* (2002), pp. 623–632.

[118] S. Suri, C. D. Toth, and Y. Zhou. Range Counting over Multidimensional Data Streams. *Proceedings of the 20th Annual Symposium on Computational Geometry*, pp. 160–169, 2004.

[119] V. Vapnik and A. Chervonenkis. On the Uniform Convergence of Relative Frequencies of Events to their Probabilities. *Theory Probab. Appl.*, 16:264–280, 1971.

[120] J. S. Vitter. Random Sampling with a Reservoir. *ACM Trans. Math. Softw.* (11) (1985), no. 1, 37–57.

[121] S. Valverde and R. Sol. Network Motifs in Computational Graphs: A Case Study in Software Architecture. *Physical Review E* (72), 2005

[122] D. J. Watts and S. H. Strogatz. Collective Dynamics of Small-World Networks. *Nature* (393), 440–442.

[123] X. Yan, P. S. Yu, and J. Han. Graph Indexing: A Frequent Structure-Based Approach *Proceedings of the SIGMOD*, 2004, pp. 335–346.

[124] T. Zhang, R. Ramakrishnan, and M. Livny. BIRCH: A New Data Clustering Algorithm and its Applications. *Journal of Data Mining and Knowledge Discovery*,1(2),pp. 141-182, 1997.

[125] J. Zhao. An Implementation of Min-Wise Independent Permutation Family. (2005), http://www.icsi.berkeley.edu/ zhao/minwise/.