Streaming-Data Algorithms For High-Quality Clustering

Liadan O'Callaghan * Nina Mishra Adam Meyerson Sudipto Guha
Rajeev Motwani

October 22, 2001

Abstract

As data gathering grows easier, and as researchers discover new ways to interpret data, streaming-data algorithms have become essential in many fields. Data stream computation precludes algorithms that require random access or large memory. In this paper, we consider the problem of clustering data streams, which is important in the analysis a variety of sources of data streams, such as routing data, telephone records, web documents, and clickstreams. We provide a new clustering algorithms with theoretical guarantees on its performance. We give empirical evidence of its superiority over the commonly-used k–Means algorithm. We then adapt our algorithm to be able to operate on data streams and experimentally demonstrate its superior performance in this context.

1 Introduction

For many recent applications, the concept of a *data stream* is more appropriate than a data set. By nature, a stored data set is an appropriate model when significant portions of the data are queried again and again, and updates are small and/or relatively infrequent. In contrast, a data stream is an appropriate model when a large volume of data is arriving continuously and it is either unnecessary or impractical to store the data in some form of memory. Data streams are also appropriate as a model of access to large data sets stored in secondary memory where performance requirements necessitate access via linear scans.

In the data stream model [17], the data points can only be accessed in the order in which they arrive. Random access to the data is not allowed; memory is assumed to be small relative to the number of points, and so only a limited amount of information can be stored. In general, algorithms operating on streams will be restricted to fairly simple calculations because of the time and space constraints. The challenge facing algorithm designers is to perform meaningful computation with these restrictions.

Some applications naturally generate data *streams* as opposed to simple data sets. Astronomers, telecommunications companies, banks, stock-market analysts, and news organizations, for example,

*Contact author; e-mail:loc@cs.stanford.edu; other authors' e-mails: nmishra@hpl.hp.com, awm@cs.stanford.edu, sudipto@research.att.com,rajeev@cs.stanford.edu.

have vast amounts of data arriving continuously. In telecommunications, for example, *call records* are generated continuously. Typically, most processing is done by examining a call record once, after which records are archived and not examined again. For example, Cortes et al. [6] report working with AT&T long distance call records, consisting of 300 million records per day for 100 million customers. There are also applications where traditional (non-streaming) data is treated as a stream due to performance constraints. For researchers mining medical or marketing data, for example, the volume of data stored on disk is so large that it is only possible to make one pass (or perhaps a very small number of passes) over the data. Research on data stream computation includes work on sampling [30], finding quantiles of a stream of points [22], and calculating the $L1$-difference of two streams [11].

A common form of data analysis in these applications involves *clustering*, i.e., partitioning the data set into subsets (clusters) such that members of the same cluster are similar and members of distinct clusters are dissimilar. Typically, a cluster is characterized by a canonical element or representative called the *cluster center*. The goal is either to determine cluster centers or to actually compute the clustered partition of the data set. This paper is concerned with the challenging problem of clustering data arriving in the form of stream. We provide a new clustering algorithm with theoretical guarantees on its performance. We give empirical evidence of its superiority over the commonly-used k–Means algorithm. We then adapt our algorithm to be able to operate on data streams and experimentally demonstrate its superior performance in this context.

In what follows, we will first describe the clustering problem in greater detail, then give a high-level overview of our results and the organization of the paper, followed a discussion of earlier related work.

The Clustering Problem There are many different variants of the clustering problem [7, 16], and literature in this field spans a large variety of application areas, including data mining [10], data compression [12], pattern recognition [7], and machine learning [27]. In our work, we will focus on following version of the problem: given an integer k and a collection N of n points in a metric space, find k medians (cluster centers) in the metric space so that each point in N is assigned to the cluster defined by the median point nearest to it. The quality of the clustering is measured by the sum of squared distances (SSQ) of data points from their assigned medians. The goal is to find a set of k medians which minimize the SSQ measure. The generalized optimization problem, in which any distance metric substitutes for the squared Euclidean distance, is known as the k–Median problem[1]. Other variants of the clustering problem may not involve centers, may employ a measure other than SSQ, and may consider special cases of a metric space such as d-dimensional Euclidean space. Since most variants are known to be NP-hard, the goal is to devise algorithms that produce solutions with near-optimal solutions in near-linear running time.

Although finding an optimal solution to the k–Median problem is known to be NP-hard, many useful heuristics, including k–Means, have been proposed. The k–Means algorithm has enjoyed considerable practical success [3], although the solution it produces is only guaranteed to be a local optimum [28]. On the other hand, in the algorithms literature, several approximation algorithms have

[1]Although squared Euclidean distance is not a metric, it obeys a relaxed triangle inequality and therefore behaves much like a metric.

been proposed for k–Median. A c-approximation algorithm is guaranteed to find a solution whose SSQ is within a factor c of the optimal SSQ, even for a worst-case input. Recently, Jain and Vazirani [20] and Charikar and Guha [5] have provided such approximation algorithms for constants $c \leq 6$. These algorithms typically run in time and space $\Omega(n^2)$ and require random access to the data. Both space requirements and the need for random access render these algorithms inapplicable to data streams. Most heuristics, including k–Means, are also infeasible for data streams because they require random access. As a result, several heuristics have been proposed for scaling clustering algorithms, for example [4, 14]. In the database literature, the BIRCH system [32] is commonly considered to provide a competitive heuristic for this problem. While these heuristics have been tested on real and synthetic datasets, there are no guarantees on their SSQ performance.

Overview of Results We will present a fast k–Median algorithm called LOCALSEARCH that uses local search techniques, give theoretical justification for its success, and present experimental results showing how it out-performs k–Means. Next, we will convert the algorithm into a streaming algorithm using the technique in [13]. This conversion will decrease the asymptotic running time, drastically cut the memory usage, and remove the random-access requirement, making the algorithm suitable for use on streams. We will also present experimental results showing how the stream algorithm performs against popular heuristics.

The rest of this paper is organized as follows. We begin in Section 2 by formally defining the stream model and the k–Median problem. Our solution for k–Median is obtained via a variant called facility location, which does not specify in advance the number of clusters desired, and instead evaluates an algorithm's performance by a combination of SSQ and the number of centers used. To our knowledge, this is the first time this approach has been used to attack the k–Median problem.

Our discussion about streaming can be found in Section 3. The streaming algorithm given in this section is shown to enjoy theoretical quality guarantees. As described later, our empirical results reveal that, on both synthetic and real streams, our theoretically sound algorithm achieves dramatically better clustering quality (SSQ) than BIRCH, although it takes longer to run. Section 3.2 describes new and critical conceptualizations that characterize which instances of facility location we will solve to obtain a k–Median solution, and also what subset of feasible facilities is sufficient to obtain an approximate solution. LOCALSEARCH is also detailed in this section.

We performed an extensive series of experiments comparing LOCALSEARCH against the k–Means algorithm, on numerous low- and high-dimensional data. The results presented in Section 4.1 uncover an interesting trade-off between the cluster quality and the running time. We found that SSQ for k-means was worse than that for LOCALSEARCH, and that LOCALSEARCH typically found near-optimum (if not the optimum) solution. Since both algorithms are randomized, we ran each one several times on each dataset; over the course of multiple runs, there was a large variance in the performance of k–Means, whereas LOCALSEARCH was consistently good. LOCALSEARCH took longer to run for each trial but for most datasets found a near-optimal answer before k–Means found an equally good solution. On many datasets, of course, k–Means never found a good solution.

We view both k-Means and BIRCH as algorithms that do a quick-and-dirty job of obtaining a solution. Finding a higher quality solution takes more time, and thus as one would expect while our algorithms require more running time, they do appear to find exceptionally good solutions.

Related Work The k-Means algorithm and BIRCH [32] are most relevant to our results. We discuss these in more detail in Section 4. Most other previous work on clustering either does not offer the scalability required for a fast streaming algorithm or does not directly optimize SSQ. We briefly review these results — a thorough treatment can be found in Han and Kinber's book [15].

Partitioning methods subdivide a dataset into k groups. One such example is the k-medoids algorithm [21] which selects k initial centers, repeatedly chooses a data point randomly, and replaces it with an existing center if there is an improvement in SSQ. k-medoids is related to the CG algorithm given in Section 3.2.1, except that CG solves the facility location variant which is more desirable since in practice one does not know the exact number of clusters k (and facility location allows as input a range of number of centers). Choosing a new medoid among all the remaining points is time-consuming; to address this problem, CLARA [21] used sampling to reduce the number of feasible centers. This technique is similar to what we propose in Theorem 4. A distinguishing feature of our approach is a careful understanding of how sample size affects clustering quality. CLARANS [26] draws a fresh sample of feasible centers before each calculation of SSQ improvement. All of the k-medoid types of approaches, including PAM, CLARA, and CLARANS, are known not to be scalable and thus are not appropriate in a streaming context.

Other examples of partitioning methods include that of Bradley et al. [4] and its subsequent improvement by Farnstorm et al. [9] which repeatedly takes k weighted centers (initially chosen randomly with weight 1) and as much data as can fit in main memory and computes a k-clustering. The new k centers so obtained are then weighted by the number of points assigned, the data in memory is discarded, and the process repeats on the remaining data. A key difference between this approach and ours is that their algorithm places higher significance on points later in the data set - we assume that our data stream is not sorted in any way. Furthermore, these approaches are not known to outperform the popular BIRCH algorithm.

Hierarchical methods decompose a dataset into a tree-like structure. HAC, the most common, treats each point as a cluster and then repeatedly merges the closest ones. If the number of clusters k is known, then merging stops when only k separate clusters remain. Hierarchical algorithms are known to suffer from the problem that hierarchical merge or split operations are irrevocable [15]. Another hierarchical technique is CURE [14], which represents a cluster by multiple points that are initially well-scattered in the cluster and then shrunk toward the cluster center by a certain fraction. HAC and CURE are designed to discover clusters of arbitrary shape and thus do not necessarily optimize SSQ.

Density-based methods define clusters as dense regions of space separated by less dense regions. Such methods typically continue to grow clusters as long as their density exceeds a specified threshold. Desnsity-based algorithms like DBSCAN [8], OPTICS [2] and DENCLUE [18] are not explicitly designed to minimize SSQ.

Grid-based methods typically quantize the space into a fixed number of cells that form a grid-like structure. Clustering is then performed on this structure. Examples of such algorithms include

STING [31], CLIQUE [1], Wave-Cluster [29], and OPTIGRID [19]. Cluster boundaries in these methods are axis-aligned hyperplanes, (no diagonal separations are allowed) and so grid-based approaches are also not designed to optimize SSQ.

2 Preliminaries

We begin by defining a stream more formally. A data stream is a finite set N of points $x_1, \ldots, x_i, \ldots, x_n$ read in increasing order of the indices i. For example, these points might be vectors in $\Re^d$. Random access to the data is not allowed. A stream algorithm is allowed to remember a small amount of information about the data it has seen so far. The performance of a stream algorithm is measured by the amount of information it retains about the data that has streamed by, as well as by the usual measures: in the case of a clustering algorithm, for example, these could be SSQ and running time.

In what follows, we give a more detailed and formal definition of the k-Median problem and of the related *facility location* problem.

Suppose we are given a set N of n objects. There is some notion of "similarity" between the objects, and the goal is to group the objects into exactly k clusters so that similar objects belong to the same cluster while dissimilar objects are in different clusters. We will assume that the objects are represented as points in a metric space whose distance function conveys similarity. The k–Median problem may now be formally defined as follows:

Definition 1 (The k–Median Problem) *We are given a set N of n data points in a metric space, a distance function $d : N \times N \to \Re^+$, and a number k. For any choice of $C = \{c_1, c_2, \ldots, c_k\} \subset N$ of k cluster centers, define a partition of N into k clusters $N_1, N_2, \ldots, N_k$ such that N_i contains all points in N that are closer to c_i than to any other center. The goal is to select C so as to minimize the sum of squared distance (SSQ) measure:*

$$SSQ(N, C) = \sum_{i=1}^{k} \sum_{x \in N_i} d(x, c_i).$$

Assuming that the points in N are drawn from a metric space means that the distance function is *reflexive* (i.e., $d(x, y) = 0$ iff $x = y$) and *symmetric* (i.e., $d(x, y) = d(y, x)$), and satisfies the *triangle inequality* (i.e., $d(x, y) + d(y, z) \geq d(x, z)$). The first property is natural for any similarity measure. The second property is valid for most commonly-used similarity measures (e.g., Euclidean distance, L_1 norm, L_∞ norm), and for other measures it holds in an approximate sense (e.g., $d(x, y) + d(y, z) \geq \frac{1}{2} d(x, z)$). An approximate triangle inequality is sufficient for our purposes and will enable us to prove bounds on the performance of our algorithms, possibly with some increase in the constant factors. As will be clear from our experiments, the actual performance of our algorithms is very close to optimal, even when the provable constants are large.

In some domains, such as real space with the Euclidean metric, we can relax the restriction that C must be a subset of N, and instead allow the medians to be chosen from some larger set.

We now turn to a closely-related problem, called *facility location*. The facility location problem is a Lagrangian relaxation of k–Median. We are again given n data points to be grouped into clusters, but here, instead of restricting the number of clusters to be at most k, we simply impose a cost for each cluster. In fact, the cost is associated with a cluster center which is called a *facility*, and the cost is viewed as a *facility cost*. The facility cost prevents a solution from having a large number of clusters, but the actual number of clusters used will depend on the relationship of the facility cost to the distances between data points. The problem may be formally stated as follows (z is the facility cost):

Definition 2 (The Facility Location Problem) *We are given a set N of n data points in a metric space, a distance function $d : N \times N \rightarrow \Re^+$, and a parameter z. For any choice of $C = \{c_1, c_2, \ldots, c_k\} \subset N$ of k cluster centers, define a partition of N into k clusters $N_1, N_2, \ldots, N_k$ such that N_i contains all points in N that are closer to c_i than to any other center. The goal is to select a value of k and a set of centers C so as to minimize the facility clustering (FC) cost function:*

$$FC(N, C) = z|C| + \sum_{i=1}^{k} \sum_{x \in N_i} \Delta(x, c_i).$$

This problem is also known to be NP-hard, and several theoretical approximation algorithms are known [20, 5].

Later we will make more precise statements about the relationship between facility location and k–Median, but we can generally characterize it via some examples. First, assume we have a particular point set on which we wish to solve facility location. If opening a facility were free, then the solution with a cluster center (facility) at each point would be optimal. Also, if it were very costly to open a facility, then the optimal solution could not afford more than a single cluster. Therefore, as facility cost increases, the number of centers tends to decrease.

3 Clustering Streaming Data

Our algorithm for clustering streaming data uses a subroutine, which, for the moment, we will call LOCALSEARCH. The explanation and motivation of this subroutine are somewhat involved and will be addressed shortly.

3.1 Streaming Algorithm

Prior clustering research has largely focused on static datasets. We consider clustering *continuously arriving* data, i.e., streams. The stream model restricts memory to be small relative to the (possibly very large) data size, so algorithms like k-means, which require random access, are not suitable.

We assume that our data actually arrives in chunks $X_1, \ldots, X_n$, each of which fits in main memory. We can turn any stream into a chunked stream by simply waiting for enough points to arrive.

The streaming algorithm is as follows. For each chunk i, we first determine whether the chunk consists mostly of a set of fewer than k points repeated over and over. If it does, then we re-represent the chunk as a weighted data set in which each distinct point appears only once, but with weight equal to its original frequency in that chunk (or, if the original points had weights, the new weight is the total weight of that point in the chunk). We cluster $X - i$ using LOCALSEARCH. We then purge memory, retaining only the k weighted cluster centers found by LOCALSEARCH on X_i; the weight of each cluster center is the sum of the weights of the members that center had according to the clustering. We apply LOCALSEARCH to the weighted centers we have retained from $X_1, \ldots, X_i$, to obtain a set of (weighted) centers for the entire stream $X_1 \cup \cdots \cup X_i$.

Algorithm STREAM

For each chunk X_i in the stream

1. If a sample of size $\geq \frac{1}{\epsilon} \log \frac{k}{\delta}$ contains fewer than k distinct points then:
   - $X_i \leftarrow$ weighted representation
2. Cluster X_i using LOCALSEARCH
3. $X' \leftarrow ik$ centers obtained from chunks 1 through i iterations of the stream, where each center c obtained by clustering X_i is weighted by the number of points in X_i assigned to c.
4. Output the k centers obtained by clustering X' using LOCALSEARCH

Algorithm STREAM is memory efficient since at the ith point of the stream it retains only $O(ik)$ points. For very long streams, retaining $O(ik)$ points may get prohibitively large. In this case, we can once again cluster the weighted ik centers to retain just k centers.

Real streams may include chunks consisting mostly of a few points repeated over and over. [2] In such a situation, clustering algorithms may speed up considerably if run on the more compact, weighted representation, because making multiple passes over a highly redundant data chunk, and repeatedly processing the same point, is wasteful. Our solution (step 1) is to represent the chunk as a weighted dataset (where the weight corresponds to the number of times the point appears). Since creating a weighted data set requires time $O(nq)$ where q is the number of distinct values, we only wish to create such a weighted dataset if the data is mostly redundant, i.e., if q is small (since, for example, if there are n distinct values, $\Omega(n^2)$ time is needed to create a weighted dataset). The following simple claim shows that with a small sample we can determine if there are more than k distinct values of weight more than ϵ.

Claim 1 *If a dataset N contains $l \geq k$ distinct values of weight at least ϵ then with high probability a sample of size $v \geq \frac{1}{\epsilon} \log \frac{k}{\delta}$ contains at least k distinct values.*

Proof: Let $p_1, \ldots, p_l$ be the l distinct values of weight at least ϵ. Then the probability our sample missed k of these distinct values (say $p_1, \ldots, p_k$) is at most: $\sum_{i=1}^{k} (1 - p_i)^v \leq k(1 - \epsilon)^v \leq \delta$ by our choice of v. □

Thus, with high probability, a weighted dataset is not created in Step 1 of Algorithm STREAM if there are at least k distinct values.

[2] We introduce such a dataset in our experiments section: most of the points in our "network intrusions" dataset represent normal computer system use, and the rest represent instances of malicious behavior. Some chunks of the stream are composed almost entirely of the "normal-use" points and therefore have very few distinct vectors.

Step 1 affects speed but not clustering quality; steps 2-4 accomplish the actual clustering, which is provably good: previously it was shown that STREAM produces a solution whose cost is at most a constant times the cost we would get by applying LOCALSEARCH directly to the entire stream (supposing it fit in main memory).

Theorem 1 *[13] If at each iteration i, a c-approximation algorithm is run in Steps 2 and 4 of Algorithm STREAM, then the centers output are a $5c$-approximation to the optimum SSQ for $X_1 \cup \cdots \cup X_i$, assuming a distance metric on R^d.* [3]

The above theorem is only intended to show that Algorithm STREAM cannot output an arbitrarily bad solution. In practice, we expect that the solution STREAM finds will be significantly better than 5 or 10 times worse than optimum. Our synthetic stream experimental results are consistent with this expectation.

3.2 LOCALSEARCH Algorithm

STREAM needs a simple, fast, constant-factor-approximation k–Median subroutine. We believe ours is the first such algorithm that also guarantees flexibility in k.

Here we will describe our new algorithm for k–Median, which is based on the concept of local search, in which we start with an initial solution and then refine it by making local improvements. Throughout this section, we will refer to the "cost" of a set of medians, meaning the FC cost from the facility location definition.

3.2.1 Previous Work on Facility Location

Assume we have a fast algorithm A that computes an n-approximation to facility location. We begin by describing a simple algorithm[4] CG for solving the facility location problem on a set N of n points in a metric space with metric (relaxed metric) $d(\cdot, \cdot)$, when facility cost is z. First, we will make one definition.

Assume that we have a feasible solution to facility location on N given $d(\cdot, \cdot)$ and z. That is, we have some set $I \subseteq N$ of currently open facilities, and an assignment for each point in N to some (not necessarily the closest) open facility. For every $x \in N$ we will define the *gain* of x to be the amount of cost we would save (or further expend) if we were to open a facility at x (if one does not already exist), and then perform all possible advantageous reassignments and facility closings, subject to the following two constraints: first, that points cannot be reassigned except to x, and second, that a facility can be closed only if its members are first reassigned to x (if it has no members, it can of course be closed). What would be the net savings if we were to make these improvements? First, we would pay z to open x if it were not already open. Next, we would certainly reassign to x every point whose current assignment distance is greater than $d(x, y)$, for a total savings of, say, t. We would then

[3] Without the Euclidean assumption, Algorithm STREAM is a $10c$-approximation.

[4] It has been shown by Charikar and Guha [5] that this algorithm will achieve a $(1 + \sqrt{2})$-approximation to facility location on N with facility cost z.

close any facilities without members; a closure of u centers would result in a savings of uz. Finally, we might find that there were still some open facilities such that closing them and reassigning their members to x would actually produce a net savings. In this case, merely performing the reassignment (without the closure) would certainly produce an expenditure rather than a savings, since otherwise these members would have been reassigned earlier. However, if the reassignment cost were lower than z, this operation would produce a savings. Here, if we were to close v centers, our savings would be vz, less the added assignment cost (call this a). $gain(x)$, then, is the net savings of these four steps: $-z + t + uz + vz - a$ (the $-z$ is omitted if x is already open).

Algorithm CG(data set N, facility cost z)

1. Run algorithm A to compute a set $I \subseteq N$ of facilities, that gives an n-approximation to facility location on N with facility cost z.
2. Repeat $\Omega(\log n)$ times:
   - Randomly order N.
   - For each x in this random order: calculate $gain(x)$, and if $gain(x) > 0$, add a facility there and perform the allowed reassignments and closures.

3.2.2 Our New Algorithm

The above algorithm does not directly solve k–Median but could be used as a subroutine to a k–Median algorithm, as follows. We first set an initial range for the facility cost z (between 0 and an easy-to-calculate upper bound); we then perform a binary search within this range to find a value of z that gives us the desired number k of facilities; for each value of z that we try, we call Algorithm CG to get a solution.

Binary Search Two questions spring to mind, however: first, will such a binary search technique work, and second, will this algorithm, which uses Algorithm CG as a subroutine, find good solutions quickly?

First, we will examine the binary search idea by trying to understand how facility cost affects the character of the optimal facility location solution. Consider a dataset N of n points, with a distance function $d(\cdot,\cdot)$, on which we wish to solve facility location. We can show that, if the facility cost increases, the number of centers in the optimal solution does not increase, the SSQ does not decrease, and the total (FC) solution cost increases. As opening a new facility becomes more expensive, we are forced to close centers and give solutions with higher assignment distances and FC cost. This proof will justify performing a binary search on z.

Theorem 2 *Assume we have a set N of points with a metric $d : N \times N \to \Re^+$. Then given facility costs f and f' with $0 \le f < f'$, a set C of k centers that is optimal for f, and a set C' of k' centers that is optimal for f', the following are true:*

1. $k \ge k'$
2. $SSQ(N, C) \le SSQ(N, C')$

3. $fk + SSQ(N,C) < f'k' + SSQ(N,C')$

Furthermore, $SSQ(N,C) = SSQ(N,C')$ *iff* $k = k'$*.*

Proof: Let $a = SSQ(N,C)$, $a' = SSQ(N,C')$. Note $a \geq 0$ and and $a' \geq 0$. Then $FC(N,C) = fk + a$ and $FC(N,C') = fk' + a'$, for facility cost f. For facility cost f', $FC(N,C) = f'k + a$ and $FC(N,C') = f'k' + a'$. By the optimality of C for f we have (I): $fk + a \leq fk' + a'$, and, by the optimality of C' for f' we have (II): $f'k + a \geq f'k' + a'$. Then we have $f(k - k') \leq f'(k - k')$. Since $f < f'$, it must be that $k \geq k'$. Since $k \geq k'$, we can see by (I) that $a \leq a'$. Combining (I) and the fact that $f < f'$, we have that $fk + a \leq f'k' + a'$.

If $a = a'$, then if $f = 0$ then $C = N$; that is, $k = n$ and $a = 0$. But then since $a = a'$, we know that C' must also be N so that $k' = n$. If $a = a'$ and $f \neq 0$ then we can combine (I) and (II) to show that $k = k'$. Therefore, $a = a' \Rightarrow k = k'$. If $k = k'$ (note $1 \leq k \leq n$), then we can again combine (I) and (II) to show that $a = a'$. Therefore, $a = a'$ iff $k = k'$. □

For a given instance of facility location, there may exist a k such that there is no facility cost for which an optimal solution has exactly k medians. However, if the dataset is "naturally k-clusterable," then our algorithm should find k centers. Indeed, if allowing k centers rather than only $k-1$ gives a significant decrease in assignment cost, then there is an f for which some optimal facility location solution has k centers. The following theorem defines the term "significant decrease:"

Theorem 3 *Given a dataset N, let A_i denote the best assignment cost achievable for an instance of FL on N if we allow i medians. If N has the property that $A_k \leq \frac{A_j}{k-j+1}$ for all $j < k$ then there is a facility location solution with k centers.*

Proof: For an FL solution to yield k centers, the FL solution for $j < k$ centers (and for $l > k$ centers) must have worse total cost than the FL solution for k centers, i.e., $fj + A_j \geq fk + A_k$ and $fl + A_l \geq fk + A_k$. In other words, when $\frac{A_j - A_k}{k-j} \leq f \leq \frac{A_k - A_l}{l-k}$. $\frac{A_k - A_l}{l-k}$ is at most A_k. So in order for some facility cost to exist we need $A_k \leq \frac{A_j - A_k}{k-j}$ for all $j > k$. The result then follows by the assumption given in the statement of the theorem. □

For example, if we realize an improvement going from $k-1$ to k centers of the form $A_k \leq \frac{A_{k-1}}{k}$ then our facility location algorithm will find k centers.

The next question to answer is whether this method of calling CG as a subroutine of a binary search gives good solutions to k–Median in a timely fashion. We have experimented extensively with this algorithm and have found that it gives excellent SSQ and often achieves k–Median solutions that are very close to optimal. In practice, however, it is prohibitively slow. If we analyze its running time, we find that each $gain(x)$ calculation takes $\Theta(n)$ time, [5] and since each of the $\log n$ iterations requires n such $gain$ computations, the overall running time for a particular value of facility cost is $\Theta(n^2 \log n)$. To solve k–Median, we would perform a binary search on the facility cost to find a cost z that gives the desired number of clusters, and so we have a $\Theta(n \log n)$-time operation at each step in the binary search. Because there may be a non-linear relationship between the facility cost and the

[5] because we must consider reassigning each point to x and closing facilities whose points would be reassigned

number of clusters in the solution we get, this binary search can require many iterations, and so the overall k–median algorithm may be slow. Furthermore, most implementations would keep the $n \times n$ distance matrix in memory to speed up the algorithm by a constant factor, making running time and memory usage $\Omega(n^2)$ – probably too large even if we are not considering the data stream model of computation. Therefore, we will describe a new local search algorithm that relies on the correctness of the above algorithm but avoids the super-quadratic running time by taking advantage of the structure of local search in certain ways.

Finding a Good Initial Solution First, we observe that CG begins with a quickly-calculated initial solution that is guaranteed only to be an n-approximation. If we begin with a better initial solution, we will require fewer iterations in step 2 above. In particular, on each iteration, we expect the cost to decrease by some constant fraction of the way to the best achievable cost [5]; if our initial solution is a constant-factor approximation rather than an n-approximation, we can reduce our number of iterations from $\Theta(\log n)$ to $\Theta(1)$. Therefore, we will begin with an initial solution found according to the following algorithm, which takes as input a dataset N of n points and a facility cost z.

Algorithm InitialSolution(data set N, facility cost z)

1. Reorder data points randomly
2. Create a cluster center at the first point
3. For every point after the first,
   - Let d be the distance from the current data point to the nearest existing cluster center
   - With probability d/z create a new cluster center at the current data point; otherwise add the current point to the best current cluster

This algorithm runs in $O(n^2)$ time. It has been shown [24] that this algorithm obtains an expected 8-approximation to optimum.

Sampling to Obtain Feasible Centers Next we present a theorem that will motivate a new way of looking at local search. Assume the points $c_1, \ldots, c_k$ constitute an optimum choice of k medians for the dataset N, that C_i is the set of points in N assigned to c_i, and that r_i is the average distance from a point in C_i to c_i for $1 \leq i \leq k$. Assume also that, for $1 \leq i \leq k$, $|C_i|/|N| \geq p$. Let $0 < \delta < 1$ be a constant and let S be a set of $m = \frac{8}{p} \log \frac{2k}{\delta}$ points drawn independently and uniformly at random from N. Then we have the following theorem regarding solutions to k–Median that are restricted to medians from S (and not merely from N at large):

Theorem 4 *With high probability, the optimum k–Median solution in which medians are constrained to be from S has cost at most 3 times the cost of the optimum unconstrained k–Median solution (where medians can be arbitrary points in N).*

Proof: With a sufficiently large sample we show that from each C_i we can obtain $4 \log \frac{2k}{\delta}$ points, of which at least one is sufficiently close to the optimum center c_i. In a sample of size $m = \frac{8}{p} \log \frac{2k}{\delta}$ the probability we obtain fewer than $mp/2$ points from a given cluster is less than $\frac{\delta}{2k}$, by Chernoff bounds. Thus by the Union bound the probability we obtain fewer than $mp/2$ points from any cluster is at most $\frac{\delta}{2}$. Given that with high probability we obtain $mp/2$ points from a cluster C_i, the chance that all points in S are far from c_i is small. In particular, the probability that no point from S is within distance $2r_i$ of the optimum center c_i is at most $\frac{1}{2}^{mp/2} \leq \frac{1}{2}^{\log\frac{2k}{\delta}} = \frac{\delta}{2k}$ by Markov's Inequality. So the probability cluster C_i or ...or C_k has no point within distance $2r_i$ is at most $\frac{\delta}{2}$, by the Union bound. Combining, we have that the probability we either fail to obtain at least $mp/2$ points per cluster or fail to find one sufficiently close to a center is at most δ.

If, for each cluster C_i, our sample contains a point x_i within $2r_i$ of c_i, the cost of the median set $\{x_1, \ldots, x_k\}$ is no more than 3 times the cost of the optimal k–Median solution (by the triangle inequality, each assignment distance would at most triple). $\square$

In some sense we are assuming that the smallest cluster is not too small. If, for example, the smallest cluster contains just one point, i.e., $p = 1/|N|$, then clearly no point can be overlooked as a feasible center. We view such a small subset of points as outliers, not clusters. Hence we assume that outliers have been removed.

This theorem leads to a new local-search paradigm: rather than searching the entire space of possible median-set choices, we will search in a restricted space that is very likely to have good solutions. If instead of evaluating $gain()$ for every point x, we only evaluate the $gain()$ of a randomly chosen set of $\Theta(\frac{1}{p}\log k)$ points, we are still likely to choose good points for evaluation (i.e., points that would make good cluster centers relative to the optimal centers), but we will finish our computation sooner. In fact, if we select a set of $\Theta(\frac{1}{p}\log k)$ points up front as feasible centers, and then only ever compute the $gain()$ for each of these points, we will have in addition the advantage that solutions for nearby values of z ought not to be too different.

Finding a Good Facility Cost Faster Third, we recall that each successive set of $gain()$ operations tends to bring us closer to the best achievable cost by a constant fraction. The total gain achieved during one set of $gain()$ computations (and the associated additions and deletions of centers) should then be smaller than the previous total gain by a constant fraction, in an absolute sense. Assume that, for the current value of z, our cost is changing very little but our current solution has far from k centers; we can abort and change the value of z, knowing that it is very unlikely that our best-possible solution for this z would have k centers. We can also stop trying to improve our solution when we get to a solution with k centers and find that our solution cost is improving by only a small amount after each new set of $gain()$ operations. If our current number of centers is far from k, our definition of "small improvement" will probably be much more broad than if we already have exactly k centers. In the first case we are simply observing that our current value of z is probably quite far from the one we will eventually want to arrive at, and we are narrowing the interval of possible z values in which we will search. In the second case, we are more or less happy with our solution, but, because the number of centers is unlikely to change for this value of z, we want to get as good a solution as possible without wasting many iterations improving by an insignificant fraction.

Therefore, we will give a Facility Location subroutine that our k–Median algorithm will call; it will take a parameter $\epsilon \in \Re$ that controls how soon it stops trying to improve its solution. The other parameters will be the data set N of size n, the metric or relaxed metric $d(\cdot, \cdot)$, the facility cost z, and an initial solution (I, a) where $I \subseteq N$ is a set of facilities and $a : N \rightarrow I$ is an assignment function.

Algorithm FL(N, $d(\cdot,\cdot)$, z, ϵ, (I,a))

1. Begin with (I, a) as the current solution
2. Let C be the cost of the current solution on N. Consider the feasible centers in random order, and for each feasible center y, if $gain(y) > 0$, perform all advantageous closures and reassignments (as per $gain$ description), to obtain a new solution (I', a') [a' should assign each point to its closest center in I']
3. Let C' be the cost of the new solution; if $C' \leq (1 - \epsilon)C$, return to step 2

Now we will give our k–Median algorithm.

Algorithm LOCALSEARCH $(N, d(\cdot,\cdot), k, \epsilon, \epsilon', \epsilon'')$

1. Read in n data points.
2. Set $z_{min} = 0$
3. Set $z_{max} = \sum_{x \in N} d(x, x_0)$, where x_0 is an arbitrary point in N
4. Set z to be $\frac{z_{max}+z_{min}}{2}$
5. Obtain an initial solution (I, a) using Algorithm InitialSolution(N, z).
6. Select $\Theta(\frac{1}{p} \log k)$ random points to serve as feasible centers
7. While more or fewer than k centers and $z_{min} < (1 - \epsilon'')z_{max}$:
   - Let (F, g) be the current solution
   - Run $FL(N, d, \epsilon, (F, g))$ to obtain a new solution (F', g')
   - If $|F'|$ is "about" k, run $FL(N, d, \epsilon', (F', g'))$ to obtain a new solution; reset (F', g') to be this new solution
   - If $|F'| > k$ then set $z_{min} = z$ and $z = \frac{z_{max}+z_{min}}{2}$; else if $|F'| > k$ then set $z_{max} = z$ and $z = \frac{z_{max}+z_{min}}{2}$
8. To simulate a continuous space, move each cluster center to the center-of-mass for its cluster
9. Return our solution (F', g')

In the third step of the while loop, "If $|F'|$ is "about" k," means that either $k - 1 \leq |F'| \leq k + 1$ or $(0.8)k \leq |F'| \leq (1.2)k$. The initial value of z_{max} is chosen as a trivial upper bound on the value of z we will be trying to find. [6] The running time of LOCALSEARCH is $O(nm + nk \log k)$ where m is the number of facilities opened by InitialSolution. m depends on the properties of the dataset but is usually small, so this running time is a significant improvement over previous algorithms.

[6] If the facility cost f is the sum of assignment costs when we open only some particular facility, then the solution that opens only this facility will have cost $2f$. There may be an equally cheap solution that opens exactly two facilities and has zero assignment cost (if every point is located exactly at a facility), and there may be cheaper solutions that open only one facility (if there is a different facility for which the sum of assignment costs would be less than f), but there cannot exist an optimal solution with more than two facilities. This value f is therefore an upper bound for z since k–Median is trivial when $k = 1$.

4 Experiments

4.1 Empirical Evaluation of LOCALSEARCH

We present the results of experiments comparing the performance of k–Means and LOCALSEARCH. First we will describe the precise implementation of k–Means used in these experiments. Then we will compare the behavior of k–Means to LOCALSEARCH on a variety of synthetic datasets.

Algorithm k–Means(data set $N \subseteq \Re^d$, integer k)

1. Pick k points $c_1, \ldots, c_k$ from $\Re^d$
2. "Assign" each point in N to the closest of the c_i. We say a point x is a "member of cluster j" if it is assigned to c_j.
3. Repeat until no point changes membership, or until some maximum number of iterations:
   - Recalculate each c_i to be the Euclidean mean of the points assigned to it
   - Reassign each point in N to the closest of the (updated) c_i.

The k initial centers are usually either chosen randomly from $\Re^d$, or else given the values of random points from N. Often, if the centers are chosen to be random points in $\Re^d$, the value of the ith coordinate of the jth center is chosen uniformly at random from the interval $[m_i, M_i]$, where m_i (M_i, respectively) is the minimum (resp. *maximum*) value of the ith coordinate of any point in N. If the second initialization method is used, each center is generally given the value of a point chosen uniformly at random from N. Both methods are widely used in practice. The first may work better in datasets in which many of the points are in a tight cluster, but a significant number of points are in less populous clusters in other parts of the space. If centers are chosen to be data points, the algorithm is very likely to pick more than one center from the tight cluster and then settle into a local optimum in which several centers represent the tight cluster leaving the rest of the clusters to be grouped incorrectly. The second method may work better in datasets that are spread out and span a region in $\Re^d$ that is mostly sparse. In all our experiments we used the first initialization method.

We conducted all experiments on a Sun Ultra 2 with two, 200MHz processors, 256 MB of RAM, and 1.23 GB of swap space,[7] running SunOS 5.6.

4.1.1 Experiments on Low-Dimensional Datasets

We generated twenty-eight small datasets and ran LOCALSEARCH and k–Means on each. To put the results of both algorithms in perspective relative to earlier work, we ran both algorithms on a dataset distributed by the authors of BIRCH [32], which consists of one hundred, two-dimensional Gaussians in a ten-by-ten grid, with some overlap.[8]

[7] Our processes never used more than one processor or went into swap.

[8] All the datasets described in this section have clusters of approximately the same volume and number of points. As noted in Theorem 4 the clusters need not be this regular for LOCALSEARCH to be able to find them. In section 4.2 we will report on experiments on a synthetic dataset of clusters of varying density, volume, and number of points; in accordance with Theorem 4, LOCALSEARCH still performs well in these experiments.

All the datasets we generated consist of uniform-density, radius-one spheres of points, with five percent random noise. The noise is uniform over the smallest d-dimensional rectangle that is aligned with the axes and contains all the spheres.

For each dataset, we calculated the SSQ of the k sphere centers on the dataset, and we used this value as an upper bound on the optimal SSQ of any set of k medians on this dataset. Since each dataset has relatively little noise (none in the case of the Gaussian dataset), the vast majority of points are in these relatively tight spheres, so the sphere centers should be very close to the optimal centers. In a few cases, LOCALSEARCH or k–Means found better solutions than the calculated "optimal." Therefore, for each dataset, we recorded the best-known SSQ for that dataset; this is simply the minimum of the best SSQ found experimentally, and the pre-calculated upper bound.

We generated three types of datasets: grid datasets, shifted-center datasets, and random-center datasets. In our grid datasets, the spheres are centered at regular or nearly regular intervals, and the spheres are separated from each other in each dimension by at least .5.

For each grid dataset, there is a shifted-grid dataset in which the centers of the spheres are those from the grid dataset, except shifted in each dimension by .25 (the direction of each shift, i.e., positive or negative, was chosen uniformly at random). Therefore, the spheres in these sets are at almost regular intervals, with a slight skew due to the random shifting.

In the random-center datasets, centers were chosen uniformly at random within some range. In a few cases, spheres overlap somewhat, but in most cases they are well-separated. These datasets are much less symmetrical than the shifted-grid sets.

Our datasets have low dimension (at most 4, and in many cases 1 or 2), few clusters (from five to sixteen), and few points (between one and seven thousand). The dataset of Gaussians has 100,000 points and 100 clusters, but it is only two-dimensional.

On each dataset, we ran k–Means ten times, to allow for different random initializations. In the "median recalculation" step (the first part of step 3, in the preceding description of k–Means), any medians without members were assigned values chosen uniformly at random as in initialization). We also ran LOCALSEARCH ten times on each dataset. In each dataset with k spheres, we asked each algorithm to find exactly k clusters. For each execution, we measured the running time, and the SSQ of the medians it found

On the grid of Gaussians, the average SSQ of k–Means solutions was 209077, and the standard deviation was 5711. For LOCALSEARCH the SSQ was 176136 and the standard deviation 337. k–Means ran, on average, for 298 s, with a standard deviation of 113 s. LOCALSEARCH ran for 1887 s on average, with a standard deviation of 836 s.

For the other datasets, we found the average SSQ for each algorithm, and normalized by division by the best-known SSQ for that set. Figure 1 shows this normalized SSQ for the shifted-center and random-center datasets[9] The error bars represent the standard deviations for each algorithm on each

[9] For want of space we omit the results for the grid datasets, which are very similar to those for the corresponding shifted-center datasets. In general, the performance of k–Means degraded as datasets got progressively less symmetric: performance on random-center sets was worse than on shifted-center, which was worse than on grid datasets. LOCALSEARCH did very well on all examples.

set (also normalized by best-known SSQ).

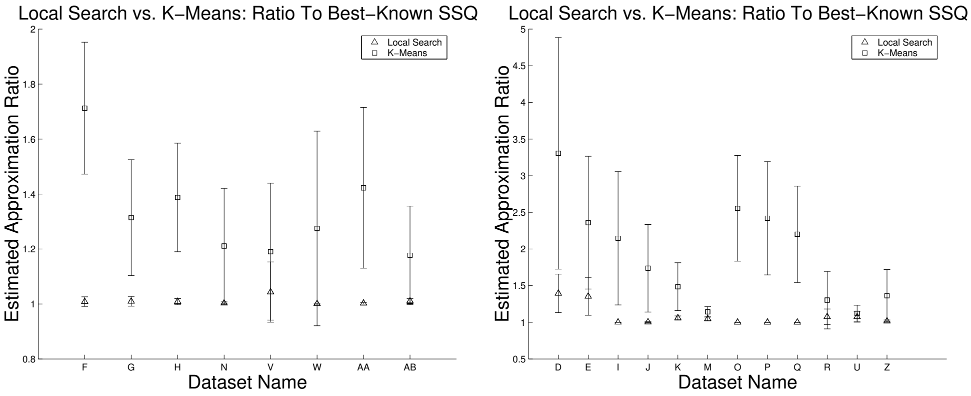

Figure 1: k–Means vs. LOCALSEARCH: Shifted-Center (left) and Random-Center Datasets

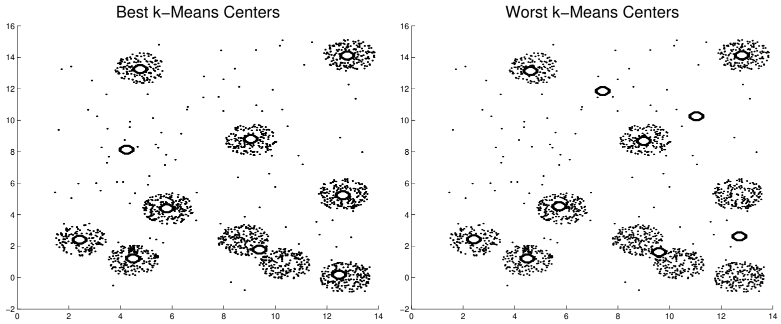

Figure 2: k–Means Centers for Dataset J

On all datasets but D and E, the average solution given by LOCALSEARCH is essentially the same as the best known. In D and E, the average SSQ is still very close, and, in fact, the best-known SSQs were among those found by LOCALSEARCH (that is, the pre-calculated "optimal" SSQs were worse). The comparatively high variance in LOCALSEARCH SSQ on these two datasets probably reflects the way they were generated; in these two sets, some of the randomly-centered spheres overlapped almost entirely, so that there were fewer "true" clusters in the dataset than we asked the algorithms to find. LOCALSEARCH had more trouble than usual with these datasets; still, it did considerably better than k–Means on both.

The behavior of k–Means was far more erratic, as the error bars show. Although the average cost of k–Means solutions on a particular dataset was usually fairly close to optimal, it was invariably worse than the average LOCALSEARCH cost, and k–Means quality varied widely.

Figures 2- 4 show this variance. Figure 2 shows the dataset J, with the best and worst medians found by k–Means (lowest- and highest-SSQ, respectively). Figure 3 shows the same for LOCALSEARCH. The LOCALSEARCH solutions barely differ, but those for k–Means are quite different. k–Means often falls into suboptimal local optima, in which one median is distracted by the random noise, forcing others to cover two spheres at the same time. LOCALSEARCH is less easily

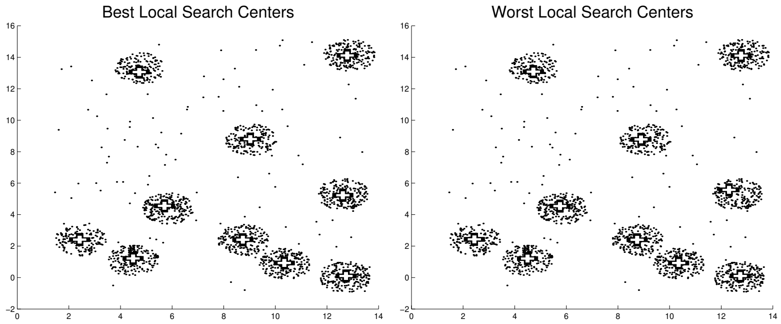

Figure 3: LOCALSEARCH Centers for Dataset J

distracted because it would rather exchange a center in the middle of noise for one in the middle of a tight cluster.

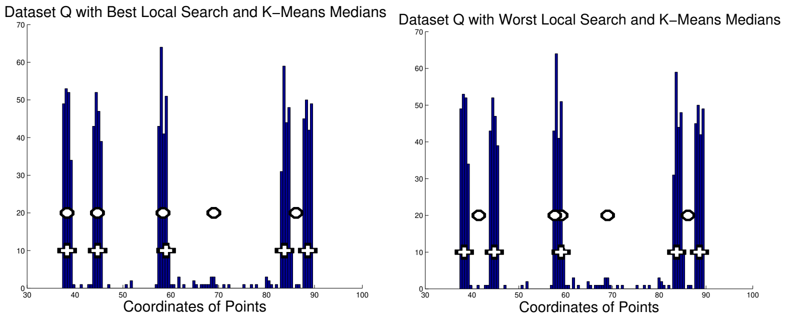

Figure 4: Histogram of Dataset Q, with k−Means and LOCALSEARCH Centers

Figure 4 plots LOCALSEARCH and k−Means centers against a one-dimensional dataset. We show the dataset as a histogram; there are five clusters, clearly visible as high spikes in the density. k−Means centers are shown by octagons whose x-coordinates give their position; similarly, LOCALSEARCH centers are displayed as "plus" signs. Here we see more examples of k−Means pitfalls: "losing" medians in the noise, covering multiple clusters with one median, and putting two medians at the same spot. By contrast, in both the best and worst case, LOCALSEARCH hits all clusters squarely (except when the noise assigned to a median pulls it slightly away), finding a near-optimal solution.

Figure 5 shows the average running times of both algorithms. Running times were generally comparable. Occasionally k−Means was much faster than LOCALSEARCH, but the slowdown was rarely more than a factor of three on average, and LOCALSEARCH never averaged more than nine seconds on any dataset. Also, the standard deviation of the LOCALSEARCH running time was high. Because LOCALSEARCH tries to adjust cluster (facility) cost until it gets the correct number of clusters, the running time can vary; the random decisions it makes can make convergence harder or easier in an unpredictable way. We should not be too concerned about LOCALSEARCH running times, however, for the following two reasons. First, LOCALSEARCH consistently found better solutions. Second, because the variance of the LOCALSEARCH solution quality was so low, it did not need to be run repeatedly before finding a very good answer; by contrast, k−Means often produced several poor answers before finding a good one. We will revisit this issue in the next section, in which we address timing and scalability issues.

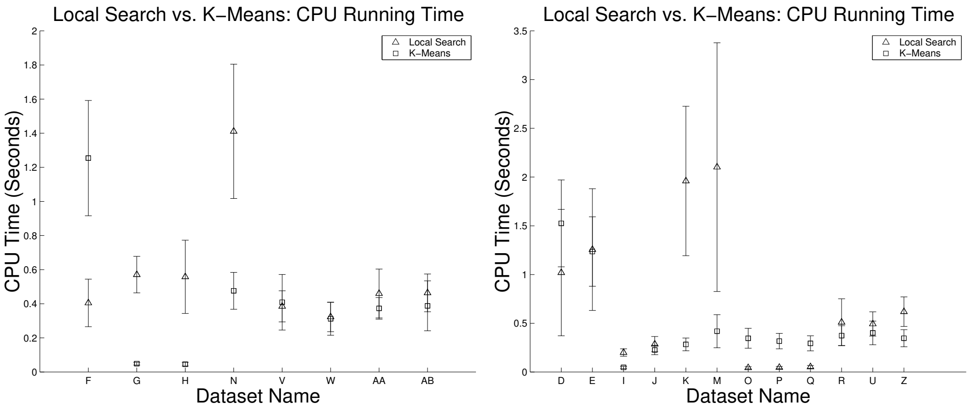

Figure 5: k–Means vs. LOCALSEARCH: Shifted-Center (left) and Random-Center Datasets

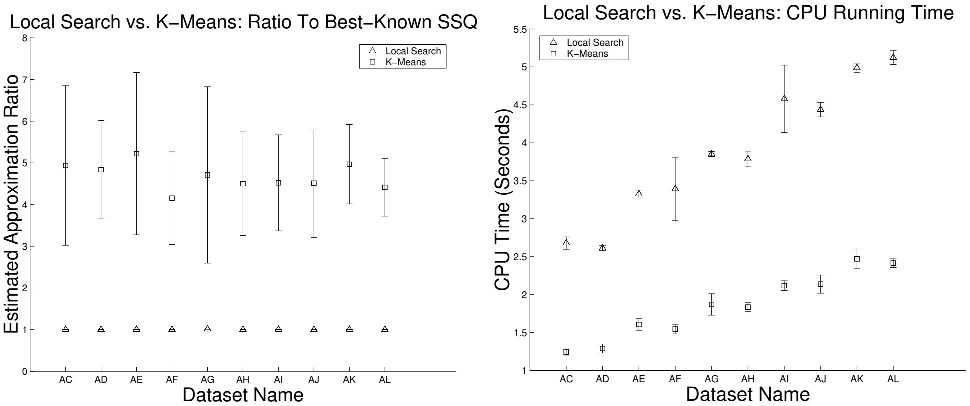

Figure 6: k–Means vs. LOCALSEARCH: High-Dimensional Datasets

These results characterize very well the differences between LOCALSEARCH and k–Means. Both algorithms make decisions based on local information, but LOCALSEARCH uses more global information as well. Because it allows itself to "trade" one or more medians for another median at a different location, it does not tend to get stuck in the local minima that plague k–Means. For this reason, the random initialization of k–Means has a much greater effect on the solution it finds, than do the random choices made by LOCALSEARCH on its solutions. This more sophisticated decision-making comes at a slight running-time cost, but, when we consider the "effective running time," i.e., the time necessary before an algorithm finds a good solution, this extra cost essentially vanishes.

4.1.2 Experiments on Small, High-Dimensional Datasets

We ran k–Means and LOCALSEARCH, ten times each, on each of ten high-dimensional datasets. All ten have one thousand points and consist of ten uniform-density, randomly-centered, d-dimensional hypercubes with edge length two, and five percent noise. The dimensionalities d of the datasets are: 100 in AC and AD, 125 in AE and AF, 150 in AG and AH, 175 in AI and AJ, and 200 in AK and AL.

As before, we ran both algorithms ten times on each dataset and averaged the SSQs of the solutions found. We slightly changed the implementation of k−Means. We initialized medians as before but changed the way we dealt with medians without members. If a median was found in some iteration to have no members, instead of setting it to a position chosen uniformly at random from the data range as we had before, we reset it to the position of a data point chosen uniformly at random. We made this choice so that memberless medians would have a chance to get members, even if most medians already had members. If a memberless median were given a random position, it would be unlikely to be close to any data point and hence unlikely to win points away from other medians; we would eventually find a clustering with fewer than k clusters. By contrast, if it were given the position of a random data point, it would have at least that point as a member, and would probably reshift the set of medians so that data points would be more evenly distributed.

Figure 6 shows the average SSQ calculated by each algorithm on each dataset, normalized by division by the best-known SSQ. The error bars represent the normalized standard deviations. For all datasets, the answer found by k−Means has, on average, four to five times the average cost of the answer found by LOCALSEARCH, which is always very close to the best-known SSQ.

The standard deviations of the k−Means costs are typically orders of magnitude larger than those of LOCALSEARCH, and they are even higher, relative to the best-known cost, than in the low-dimensional dataset experiments. This increased unpredictability may indicate that k−Means is more sensitive than LOCALSEARCHto dimensionality. Most likely, the initialization of k−Means, which is vital to the quality of solution it obtains, is more variable in higher dimensions, leading to more variable solution quality.

Figure 6 also shows the running times of these experiments; here, the differences in timing are more pronounced. LOCALSEARCH is consistently slower, although its running time has very low variance. LOCALSEARCH appears to run approximately three times as long as k−Means, although the improvement in quality makes up for the longer running time. As before, if we count only the amount of time it takes each algorithm to find a good answer, LOCALSEARCH is competitive in running time and excels in solution quality.

4.2 Clustering Streams

STREAM K-means Theorem 1 only guarantees that the performance of STREAM is boundable if a constant factor approximation algorithm is run in steps 2 and 4 of STREAM. Despite the fact that k-means has no such guarantees, due to its popularity we did experiment with running k-means as the clustering algorithm in Steps 2 and 4. Some modifications were needed to k-means to make the algorithm work in the context of STREAM and in real data.

The first modification was to make k-means work with weighted point sets. The computation of a new mean is the only place where weights need to be taken into account. If points $p_1, \ldots, p_j$ are assigned to one cluster then the usual k-means algorithm creates a cluster center at the mean of the cluster, namely $\frac{1}{j}\sum_{i=1}^{j} p_i$. The introduction of weights $w_1, \ldots, w_j$ for these points shifts the location of the center to points with higher weights: [10] $\frac{\sum_{i=1}^{j} w_i p_i}{\sum_{i=1}^{j} w_i}$.

The second modification involves a problem previously reported in the literature [3], namely the "empty cluster problem", i.e., when k-means returns fewer than k centers. Recall that the two phases of the k-means algorithm involve (1) assigning points to nearest centers and (2) computing new means. Sometimes in (1), no points are assigned to a center and as a result, k-means loses a cluster. In our implementation, if we lost a cluster, we would randomly choose a point in the space as our new cluster center. This approach worked well in our synthetic experiments. However, the KDD'99 dataset is a very sparse high dimensional space. Thus, with high probability, a random point in space is not close to any of the points in the dataset. To get around this problem, when we lose a center we randomly choose a point in the dataset.

Our experiments compare the performance of STREAM LOCALSEARCH and STREAM K-Means with BIRCH. For the sake of completeness, we give a quick overview of BIRCH.

BIRCH compresses a large dataset into a smaller one via a CFtree (clustering feature tree). Each leaf of this tree captures sufficient statistics (namely the first and second moments) of a subset of points. Internal nodes capture sufficient statistics of the leaves below them. The algorithm for computing a CFtree tree repeatedly inserts points into the leaves provided that the radius of the set of points associated with a leaf does not exceed a certain threshold. If the threshold is exceeded then a new leaf is created and the tree is appropriately balanced. If the tree does not fit in main memory then a new threshold is used to create a smaller tree. Numerous heuristics are employed to make such decisions, refer to [32] for details. Once the cftree is constructed, the weighted points in the leaves of the tree can be passed to any clustering algorithm to obtain k centers. To use BIRCH on a (chunked) stream, we used the cftree generated in iteration $(i-1)$ as a starting point for the cftree's construction in iteration i.

STREAM and BIRCH have a common method of attack: (A) Precluster and (B) Cluster. For STREAM, preclustering uses clustering itself and for BIRCH, preclustering involves creating a CFtree. This similarity enables us to directly compare the two preclustering methods. We next give experimental results on both synthetic and real KDD Cup data by comparing the two preclustering methods at each chunk of the stream. To put the results on equal footing, we gave both algorithms the same amount of space for retaining information about the stream. Hence the results compare SSQ and running time.

Synthetic Data Stream We generated a stream approximately 16MB in size, consisting of 50,000 points in 40-dimensional Euclidean space. The stream was generated similarly to the datasets described in 4.1.2, except that the diameter of clusters varied by a factor of 9, and the number of points by a factor of 6.33. We divided the point set into four consecutive chunks, each of size 4MB, and

[10] This modification is equivalent to running k-means on the larger multiset of points where each point p_i is repeated w_i times.

calculated an upper bound on the SSQ for each, as in previous experiments, by finding the SSQ for the centers used to generate the set. We ran experiments on each of the four "prefixes" induced by this segmentation into chunks: the prefix consisting of the first chunk alone, that consisting of the second chunk appended after the first, the prefix consisting of the concatenation of the first three chunks, and the prefix consisting of the entire stream.

On each prefix, we ran BIRCH once to generate the BIRCH CF-tree, and then ran BIRCH followed by Hierarchical Agglomerative Clustering (HAC) on the CF-tree. We then applied our streamed LOCALSEARCH algorithm to each prefix. To compare the quality obtained by our streamed LOCALSEARCH algorithm to that obtained by other popular algorithms, we next ran LOCALSEARCH on the CF-tree; these experiments show that LOCALSEARCH does a better job than BIRCH of pre-clustering this data set. Finally, to examine the general applicability of our streaming method, we ran streamed k–Means, and also ran k–Means on the CF-tree; we again found that the streaming method gave better results, although not as good as the streamed LOCALSEARCH algorithm. The BIRCH-HAC combination, which we ran only for comparison, gave the poorest clustering quality.

As in previous experiments, we ran LOCALSEARCH and k–Means ten times each on any dataset (or CF-tree) on which we tested them. Since BIRCH and HAC are not randomized, this repetition was not necessary when we generated CF-trees or ran HAC.

In the left-hand-side chart of figure 7, we show the average clustering quality (in SSQ) achieved by streamed LOCALSEARCH, alongside the average SSQ that LOCALSEARCH acheived when applied to the CF-tree. We compare both with the best-known value, which is calculated as in Section 4.1. LOCALSEARCH as a streaming algorithm consistently bested the BIRCH-LOCALSEARCH combination and, for every prefix, the best-known solution was a solution found by streamed LOCALSEARCH. In particular, it outperformed the BIRCH-LOCALSEARCH algorithm by a factor of 3 or 4 in SSQ. The reason for this performance gap was that the BIRCH CF-trees usually had one "mega-center," a point of very high weight, along with a few points of very small weight; in other words, BIRCH missummarized the data stream by grouping a large number of points together at their mean. Because of the way we generated the stream, any set of points that is large relative to our cluster size must include points from many (far-apart) clusters. Summarizing such a group of points as one point introduces error, because the next level of computation cannot separate these points, and is forced to give a high-SSQ answer.

The right-hand-side chart in figure 7 compares the average clustering quality of streamed k–Means and the BIRCH-k–Means combination. As before, the streaming method outperformed the CF-tree summarization method, although less dramatically (the SSQ achieved by streamed k–Means is lower by a factor of somewhat more than 2). Because we were running k–Means on the same CF-trees as those input to LOCALSEARCH, these results again reflected the incorrect summarizations of the data by the CF-tree. Still, k–Means was less able than LOCALSEARCH to handle this high-dimensional data set, and so the summarization into initial clusters was not as strong as that found by LOCALSEARCH; consequently, the difference in the final clusterings of streamed k–Means and BIRCH/k–Means was less dramatic.

For both k–Means and LOCALSEARCH, the benefits in SSQ came at an almost equal cost in terms of running time: streamed LOCALSEARCH was between 3 and 4 times slower than BIRCH/LOCALSEARCH, and streamed k–Means was approximately twice as slow as BIRCH/k–Means.

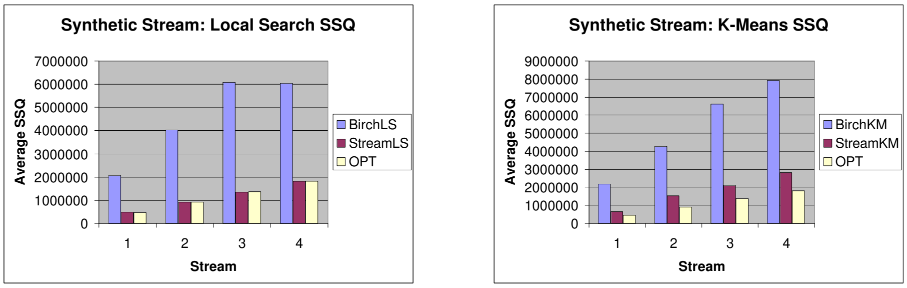

Figure 7: BIRCH vs. STREAM SSQ: LOCALSEARCH and K-Means

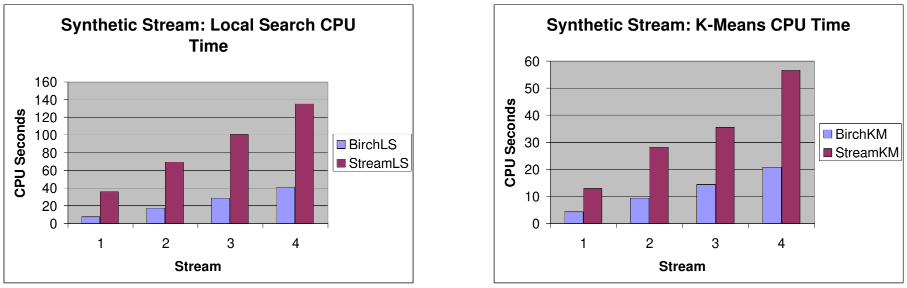

Figure 8: BIRCH vs. STREAM CPU Time: LOCALSEARCH and K-Means

Network Intrusions Recent trends in e-commerce have created a need for corporations to protect themselves from cyber attacks. Manually detecting such attacks is a daunting task given the quantity of network data. As a result many new algorithms have been proposed for automatically detecting such attacks. Clustering, and in particular algorithms that minimize SSQ, are popular techniques for detecting intrusions [25, 23]. Since detecting intrusions the moment they happen is tantamount to protecting a network from attack, intrusions are a particularly fitting application of streaming. Offline algorithms simply do not offer the immediacy required for successful network protection.

In our experiments we used the KDD-CUP'99[11] intrusion detection dataset which consists of two weeks of raw TCP dump data for a local area network simulating a true Air Force environment with occassional attacks. Features collected for each connection include the duration of the connection, the number of bytes transmitted from source to destination (and vice versa), the number of failed login attempts, etc. All 34 continuous attributes out of the total 42 attributes available were selected for

[11] http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html

clustering. One outlier point was removed. The dataset was treated as a stream of nine 16-Mbyte sized chunks. The data was clustered into five clusters since there were four types of possible attacks (plus no attack). The four attacks included denial of service, unauthorized access from a remote machine (e.g., guessing password), unauthorized access to root, and probing (e.g., port scanning).

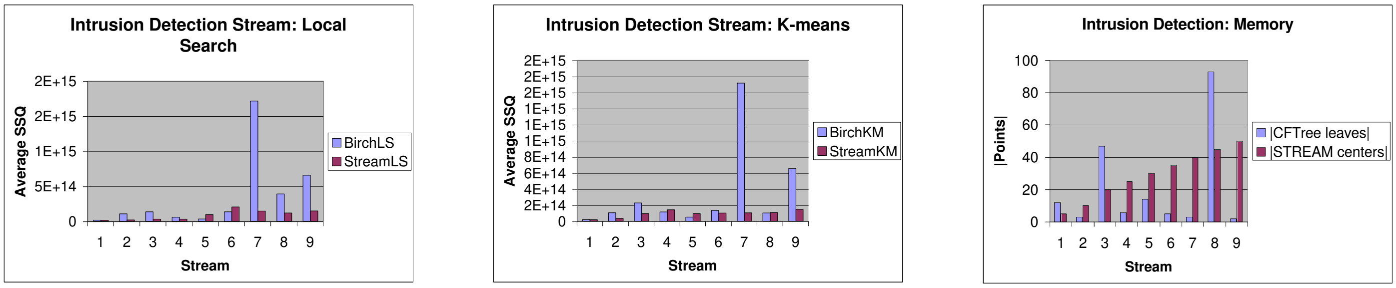

Figure 9: Left: BIRCH vs. STREAM SSQ: LOCALSEARCH and K-Means; Right: Memory Usage

The leftmost chart in Figure 9 compares the SSQ of BIRCH-LS with that of STREAMLS; the middle chart makes the same comparison for BIRCH k–Means and STREAM k–Means. BIRCH's performance on the 7th an 9th chunks can be explained by the number of leaves in BIRCH's CFTree, which appears in the third chart.

Even though STREAM and BIRCH are given the same amount of memory, BIRCH does not fully take advantage of it. BIRCH's CFTree has 3 and 2 leaves respectively even though it was allowed 40 and 50 leaves, respectively. We believe that the source of the problem lies in BIRCH's global decision to increase the radius of points allowed in a leaf when the CFTree size exceeds constraints. For many datasets BIRCH's decision to increase the radius is probably a good one - it certainly reduces the size of the tree. However, this global decision can fuse points from separate clusters into the same CF leaf. Running any clustering algorithm on the fused leaves will yield poor clustering quality; here, the effects are dramatic.

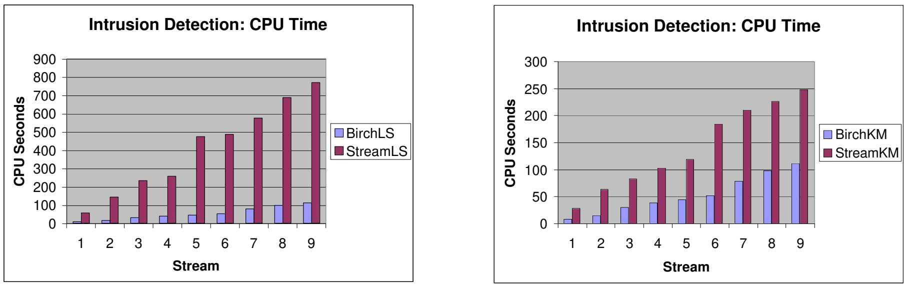

Figure 10: BIRCH vs. STREAM CPU Time: LOCALSEARCH and K-Means

In terms of cumulative average running time, Figure 10, BIRCH is faster. STREAM LS varies in its running time due to the creation of a weighted dataset (Step 1). On the fourth and sixth chunks of the stream, the algorithm always created a weighted dataset, hence the fast (not shown, but twelve second) running time. While our algorithm never generated a weighted dataset when it shouldn't (as enforced by Claim 1), there was one time when it should have generated a weighted dataset but didn't. On the fifth chunk, the algorithm created a weighted dataset on nine out of its ten runs. Thus on nine out of ten runs, the algorithm ran very quickly. On the tenth run however, the algorithm performed so many gain calculations that the CPU performance sharply degraded, causing the average running time to be very slow.

Overall, our results point to a cluster quality vs. running time tradeoff. In applications where speed is of the essence, e.g., clustering web search results, BIRCH appears to do a reasonable quick and dirty job. In applications like intrusion detection or target marketing where mistakes can be costly our STREAM algorithm exhibits superior SSQ performance.

References

[1] R. Agrawal, J.E. Gehrke, D. Gunopulos, and P. Raghavan. Automatic subspace clustering of high dimensional data for data mining applications. In *Proc. SIGMOD*, pages 94–105, 1998.

[2] M. Ankerst, M. Breunig, H. Kriegel, and J. Sander. Optics: Ordering points to identify the clustering structure. In *Proc. SIGMOD*, 1999.

[3] P.S. Bradley and U.M. Fayyad. Refining initial points for K-Means clustering. In *Proc. 15th Intl. Conf. on Machine Learning*, pages 91–99, 1998.

[4] P.S. Bradley, U.M. Fayyad, and C. Reina. Scaling clustering algorithms to large databases. In *Proc. KDD*, pages 9–15, 1998.

[5] M. Charikar and S. Guha. Improved combinatorial algorithms for the facility location and k-median problems. In *Proc. FOCS*, 1999.

[6] C. Cortes, K. Fisher, D. Pregibon, and A. Rogers. Hancock: A language for extracting signatures from data streams. In *Proc. of the 2000 ACM SIGKDD Intl. Conf. on Knowledge and Data Mining*, pages 9–17, August 2000.

[7] R.O. Duda and P.E. Hart. *Pattern Classification and Scene Analysis.* Wiley, 1972.

[8] M. Ester, H. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases, 1996.

[9] F. Farnstrom, J. Lewis, and C. Elkan. True scalability for clustering algorithms. In *SIGKDD Explorations*, 2000.

[10] U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors. *Advances in Knowledge Discovery and Data Mining.* MII Press, Mento Park, 1996.

[11] J. Feigenbaum, S. Kannan, M. Strauss, and M. Viswanathan. An approximate l1-difference algorithm for massive data streams, 1999.

[12] A. Gersho and R.M. Gray. *Vector Quantization and Signal Compression.* Kluwer Academic Publishers, Norwell, MA, 1992.

[13] S. Guha, N. Mishra, R. Motwani, and L. O'Callaghan. Clustering data streams. In *Proc. FOCS*, pages 359–366, 2000.

[14] S. Guha, R. Rastogi, and K. Shim. CURE: An efficient clustering algorithm for large databases. In *Proc. SIGMOD*, pages 73–84, 1998.

[15] J. Han and M. Kamber, editors. *Data Mining: Concepts and Techniques.* Morgan Kaufman, 200.

[16] J. A. Hartigan. *Clustering Algorithms.* John Wiley and Sons, New York, 1975.

[17] M. Henzinger, P. Raghavan, and S. Rajagopalan. Computing on data streams, 1998.

[18] A. Hinneburg and D. Keim. An efficient approach to clustering large multimedia databases with noise. In *KDD*, 1998.

[19] A. Hinneburg and D. Keim. Optimal grid-clustering: Towards breaking the curse of dimensionality in high-dimensional clustering. In *Proc. VLDB*, 1999.

[20] N. Jain and V.V. Vazirani. Primal-dual approximation algorithms for metric facility location and k-median problems. In *Proc. FOCS*, 1999.

[21] L. Kaufman and P.J. Rousseeuw. *Finding Groups in Data. An Introduction to Cluster Analysis.* Wiley, New York, 1990.

[22] G. Manku, S. Rajagopalan, and B. Lindsley. Approximate medians and other quantiles in one pass and with limited memory, 1998.

[23] D. Marchette. A statistical method for profiling network traffic. In *Proceedings of the Workshop on Intrusion Detection and Network Monitoring*, 1999.

[24] A. Meyerson. Online facility location. *In preparation*, 2001.

[25] K. Nauta and F. Lieble. Offline network intrusion detection: Looking for footprints. In *SAS White Paper*, 2000.

[26] R.T. Ng and J. Han. Efficient and effective clustering methods for spatial data mining. In *Proc. VLDB*, pages 144–155, 1994.

[27] L. Pitt and R.E. Reinke. Criteria for polynomial-time (conceptual) clustering. *Machine Learning*, 2:371, 1987.

[28] S.Z. Selim and M.A. Ismail. K–means type algorithms: A generalized convergence theorem and characterization of local optimality. *IEEE Trans. PAMI*, 6:81–86, 1984.

[29] G. Sheikholeslami, S. Chatterjee, and A. Zhang. Wavecluster: A multi-resolution clustering approach for very large spatial databases. In *Proc. VLDB*, pages 428–439, 1998.

[30] J. Vitter. Random sampling with a reservoir, 1985.

[31] W. Wang, J. Yang, and R. Muntz. Sting: A statistical information grid approach to spatial data mining, 1997.

[32] T. Zhang, R. Ramakrishnan, and M. Livny. BIRCH: an efficient data clustering method for very large databases. In *Proc. SIGMOD*, pages 103–114, 1996.