

An Efficient k -Means Clustering Algorithm: Analysis and Implementation

Tapas Kanungo, *Senior Member, IEEE*, David M. Mount, *Member, IEEE*,
 Nathan S. Netanyahu, *Member, IEEE*, Christine D. Piatko, Ruth Silverman, and
 Angela Y. Wu, *Senior Member, IEEE*

Abstract—In k -means clustering, we are given a set of n data points in d -dimensional space \mathbf{R}^d and an integer k and the problem is to determine a set of k points in \mathbf{R}^d , called centers, so as to minimize the mean squared distance from each data point to its nearest center. A popular heuristic for k -means clustering is Lloyd's algorithm. In this paper, we present a simple and efficient implementation of Lloyd's k -means clustering algorithm, which we call the filtering algorithm. This algorithm is easy to implement, requiring a kd-tree as the only major data structure. We establish the practical efficiency of the filtering algorithm in two ways. First, we present a data-sensitive analysis of the algorithm's running time, which shows that the algorithm runs faster as the separation between clusters increases. Second, we present a number of empirical studies both on synthetically generated data and on real data sets from applications in color quantization, data compression, and image segmentation.

Index Terms—Pattern recognition, machine learning, data mining, k -means clustering, nearest-neighbor searching, k-d tree, computational geometry, knowledge discovery.

1 INTRODUCTION

CLUSTERING problems arise in many different applications, such as data mining and knowledge discovery [19], data compression and vector quantization [24], and pattern recognition and pattern classification [16]. The notion of what constitutes a good cluster depends on the application and there are many methods for finding clusters subject to various criteria, both ad hoc and systematic. These include approaches based on splitting and merging such as ISODATA [6], [28], randomized approaches such as CLARA [34], CLARANS [44], methods based on neural nets [35], and methods designed to scale to large databases, including DBSCAN [17], BIRCH [50], and ScaleKM [10]. For further information on clustering and clustering algorithms, see [34], [11], [28], [30], [29].

Among clustering formulations that are based on minimizing a formal objective function, perhaps the most widely used and studied is k -means clustering. Given a set of n data points in real d -dimensional space, \mathbf{R}^d , and an

integer k , the problem is to determine a set of k points in \mathbf{R}^d , called *centers*, so as to minimize the mean squared distance from each data point to its nearest center. This measure is often called the *squared-error distortion* [28], [24] and this type of clustering falls into the general category of variance-based clustering [27], [26].

Clustering based on k -means is closely related to a number of other clustering and location problems. These include the Euclidean *k -medians* (or the *multisource Weber problem*) [3], [36] in which the objective is to minimize the sum of distances to the nearest center and the geometric *k -center* problem [1] in which the objective is to minimize the maximum distance from every point to its closest center. There are no efficient solutions known to any of these problems and some formulations are NP-hard [23]. An asymptotically efficient approximation for the k -means clustering problem has been presented by Matousek [41], but the large constant factors suggest that it is not a good candidate for practical implementation.

One of the most popular heuristics for solving the k -means problem is based on a simple iterative scheme for finding a locally minimal solution. This algorithm is often called the *k -means algorithm* [21], [38]. There are a number of variants to this algorithm, so, to clarify which version we are using, we will refer to it as *Lloyd's algorithm*. (More accurately, it should be called the *generalized Lloyd's algorithm* since Lloyd's original result was for scalar data [37].)

Lloyd's algorithm is based on the simple observation that the optimal placement of a center is at the centroid of the associated cluster (see [18], [15]). Given any set of k centers Z , for each center $z \in Z$, let $V(z)$ denote its *neighborhood*, that is, the set of data points for which z is the nearest neighbor. In geometric terminology, $V(z)$ is the set of data points lying in the Voronoi cell of z [48]. Each stage of Lloyd's algorithm moves every center point z to the centroid of $V(z)$ and then updates $V(z)$ by recomputing the distance from each point to

- T. Kanungo is with the IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120. E-mail: kanungo@almaden.ibm.com.
- D.M. Mount is with the Department of Computer Science, University of Maryland, College Park, MD 20742. E-mail: mount@cs.umd.edu.
- N.S. Netanyahu is with the Department of Mathematics and Computer Science, Bar-Ilan University, Ramat-Gan, Israel. E-mail: nathan@cs.biu.ac.il.
- C.D. Piatko is with the Applied Physics Laboratory, The John Hopkins University, Laurel, MD 20723. E-mail: christine.piatko@jhuapl.edu.
- R. Silverman is with the Center for Automation Research, University of Maryland, College Park, MD 20742. E-mail: ruth@cfar.umd.edu.
- A.Y. Wu is with the Department of Computer Science and Information Systems, American University, Washington, DC 20016. E-mail: awu@american.edu.

Manuscript received 1 Mar. 2000; revised 6 Mar. 2001; accepted 24 Oct. 2001.

Recommended for acceptance by C. Brodley.

For information on obtaining reprints of this article, please send e-mail to: tpami@computer.org, and reference IEEECS Log Number 111599.

its nearest center. These steps are repeated until some convergence condition is met. See Faber [18] for descriptions of other variants of this algorithm. For points in general position (in particular, if no data point is equidistant from two centers), the algorithm will eventually converge to a point that is a local minimum for the distortion. However, the result is not necessarily a global minimum. See [8], [40], [47], [49] for further discussion of its statistical and convergence properties. Lloyd's algorithm assumes that the data are memory resident. Bradley et al. [10] have shown how to scale k -means clustering to very large data sets through sampling and pruning. Note that Lloyd's algorithm does not specify the initial placement of centers. See Bradley and Fayyad [9], for example, for further discussion of this issue.

Because of its simplicity and flexibility, Lloyd's algorithm is very popular in statistical analysis. In particular, given any other clustering algorithm, Lloyd's algorithm can be applied as a postprocessing stage to improve the final distortion. As we shall see in our experiments, this can result in significant improvements. However, a straightforward implementation of Lloyd's algorithm can be quite slow. This is principally due to the cost of computing nearest neighbors.

In this paper, we present a simple and efficient implementation of Lloyd's algorithm, which we call the *filtering algorithm*. This algorithm begins by storing the data points in a kd-tree [7]. Recall that, in each stage of Lloyd's algorithm, the nearest center to each data point is computed and each center is moved to the centroid of the associated neighbors. The idea is to maintain, for each node of the tree, a subset of *candidate centers*. The candidates for each node are pruned, or "filtered," as they are propagated to the node's children. Since the kd-tree is computed for the data points rather than for the centers, there is no need to update this structure with each stage of Lloyd's algorithm. Also, since there are typically many more data points than centers, there are greater economies of scale to be realized. Note that this is not a new clustering method, but simply an efficient implementation of Lloyd's k -means algorithm.

The idea of storing the data points in a kd-tree in clustering was considered by Moore [42] in the context of estimating the parameters of a mixture of Gaussian clusters. He gave an efficient implementation of the well-known EM algorithm. The application of this idea to k -means was discovered independently by Alsabti et al. [2], Pelleg and Moore [45], [46] (who called their version the blacklisting algorithm), and Kanungo et al. [31]. The purpose of this paper is to present a more detailed analysis of this algorithm. In particular, we present a theorem that quantifies the algorithm's efficiency when the data are naturally clustered and we present a detailed series of experiments designed to advance the understanding of the algorithm's performance.

In Section 3, we present a *data-sensitive* analysis which shows that, as the separation between clusters increases, the algorithm runs more efficiently. We have also performed a number of empirical studies, both on synthetically generated data and on real data used in applications ranging from color quantization to data compression to image segmentation. These studies, as well as a comparison we ran against the popular clustering scheme, BIRCH¹ [50], are reported in Section 4. Our experiments show that the filtering algorithm is quite efficient even when the clusters are not well-separated.

1. Balanced Iterative Reducing and Clustering using Hierarchies.

2 THE FILTERING ALGORITHM

In this section, we describe the filtering algorithm. As mentioned earlier, the algorithm is based on storing the multidimensional data points in a kd-tree [7]. For completeness, we summarize the basic elements of this data structure. Define a *box* to be an axis-aligned hyper-rectangle. The *bounding box* of a point set is the smallest box containing all the points. A kd-tree is a binary tree, which represents a hierarchical subdivision of the point set's bounding box using axis aligned splitting hyperplanes. Each node of the kd-tree is associated with a closed box, called a *cell*. The root's cell is the bounding box of the point set. If the cell contains at most one point (or, more generally, fewer than some small constant), then it is declared to be a *leaf*. Otherwise, it is split into two hyperrectangles by an axis-orthogonal hyperplane. The points of the cell are then partitioned to one side or the other of this hyperplane. (Points lying on the hyperplane can be placed on either side.) The resulting subcells are the *children* of the original cell, thus leading to a binary tree structure. There are a number of ways to select the splitting hyperplane. One simple way is to split orthogonally to the longest side of the cell through the median coordinate of the associated points [7]. Given n points, this produces a tree with $O(n)$ nodes and $O(\log n)$ depth.

We begin by computing a kd-tree for the given data points. For each internal node u in the tree, we compute the number of associated data points $u.count$ and weighted centroid $u.wgtCent$, which is defined to be the vector sum of all the associated points. The actual centroid is just $u.wgtCent/u.count$. It is easy to modify the kd-tree construction to compute this additional information in the same space and time bounds given above. The initial centers can be chosen by any method desired. (Lloyd's algorithm does not specify how they are to be selected. A common method is to sample the centers at random from the data points.) Recall that, for each stage of Lloyd's algorithm, for each of the k centers, we need to compute the centroid of the set of data points for which this center is closest. We then move this center to the computed centroid and proceed to the next stage.

For each node of the kd-tree, we maintain a set of *candidate centers*. This is defined to be a subset of center points that might serve as the nearest neighbor for some point lying within the associated cell. The candidate centers for the root consist of all k centers. We then propagate candidates down the tree as follows: For each node u , let C denote its cell and let Z denote its candidate set. First, compute the candidate $z^* \in Z$ that is closest to the midpoint of C . Then, for each of the remaining candidates $z \in Z \setminus \{z^*\}$, if no part of C is closer to z than it is to z^* , we can infer that z is not the nearest center to any data point associated with u and, hence, we can prune, or "filter," z from the list of candidates. If u is associated with a single candidate (which must be z^*) then z^* is the nearest neighbor of all its data points. We can assign them to z^* by adding the associated weighted centroid and counts to z^* . Otherwise, if u is an internal node, we recurse on its children. If u is a leaf node, we compute the distances from its associated data point to all the candidates in Z and assign the data point to its nearest center. (See Fig. 1.)

It remains to describe how to determine whether there is any part of cell C that is closer to candidate z than to z^* . Let H be the hyperplane bisecting the line segment $\overline{zz^*}$. (See Fig. 2.) H defines two halfspaces; one that is closer to z and

```

Filter(kdNode u, CandidateSet Z) {
    C ← u.cell;
    if (u is a leaf) {
         $z^*$  ← the closest point in Z to u.point;
         $z^*.wgtCent \leftarrow z^*.wgtCent + u.point;$ 
         $z^*.count \leftarrow z^*.count + 1;$ 
    }
    else {
         $z^*$  ← the closest point in Z to C's midpoint;
        for each ( $z \in Z \setminus \{z^*\}$ )
            if ( $z.isFarther(z^*, C)$ )  $Z \leftarrow Z \setminus \{z\}$ ;
        if ( $|Z| = 1$ ) {
             $z^*.wgtCent \leftarrow z^*.wgtCent + u.wgtCent;$ 
             $z^*.count \leftarrow z^*.count + u.count;$ 
        }
        else {
            Filter(u.left, Z);
            Filter(u.right, Z);
        }
    }
}

```

Fig. 1. The filtering algorithm.

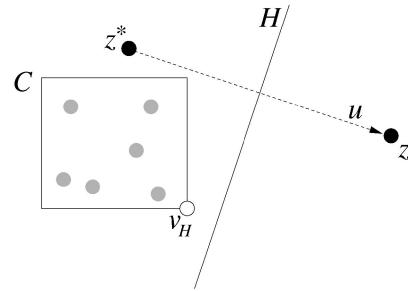
the other to z^* . If C lies entirely to one side of H , then it must lie on the side that is closer to z^* (since C 's midpoint is closer to z^*) and so z may be pruned. To determine which is the case, consider the vector $\vec{u} = z - z^*$, directed from z^* to z . Let $v(H)$ denote the vertex of C that is extreme in this direction, that is, the vertex of C that maximizes the dot product $(v(H) \cdot \vec{u})$. Thus, z is pruned if and only if $dist(z, v(H)) \geq dist(z^*, v(H))$. (Squared distances may be used to avoid taking square roots.) To compute $v(H)$, let $[C_i^{\min}, C_i^{\max}]$ denote the projection of C onto the i th coordinate axis. We take the i th coordinate of $v(H)$ to be C_i^{\min} if the i th coordinate of \vec{u} is negative and C_i^{\max} otherwise. This computation is implemented by a procedure $z.isFarther(z^*, C)$, which returns true if every part of C is farther from z than to z^* .

The initial call is to $\text{Filter}(r, Z_0)$, where r is the root of the tree and Z_0 is the current set of centers. On termination, center z is moved to the centroid of its associated points, that is, $z \leftarrow z.wgtCent/z.count$.

Our implementation differs somewhat from those of Alsabti et al. [2] and Pelleg and Moore [45]. Alsabti et al.'s implementation of the filtering algorithm uses a less effective pruning method based on computing the minimum and maximum distances to each cell, as opposed to the bisecting hyperplane criterion. Pelleg and Moore's implementation uses the bisecting hyperplane, but they define z^* (called the *owner*) to be the candidate that minimizes the distance to the cell rather than the midpoint of the cell. Our approach has the advantage that if two candidates lie within the cell, it will select the candidate that is closer to the cell's midpoint.

3 DATA SENSITIVE ANALYSIS

In this section, we present an analysis of the time spent in each stage of the filtering algorithm. Traditional worst-case analysis is not really appropriate here since, in principle, the algorithm might encounter scenarios in which it

Fig. 2. Candidate z is pruned because C lies entirely on one side of the bisecting hyperplane H .

degenerates to brute-force search. This happens, for example, if the center points are all located on a unit sphere centered at the origin and the data points are clustered tightly around the origin. Because the centers are nearly equidistant to any subset of data points, very little pruning takes place and the algorithm degenerates to a brute force $O(kn)$ search. Of course, this is an absurdly contrived scenario. In this section, we will analyze the algorithm's running time not only as a function of k and n , but as a function of the degree to which the data set consists of well-separated clusters. This sort of approach has been applied recently by Dasgupta [13] and Dasgupta and Shulman [14] in the context of clustering data sets that are generated by mixtures of Gaussians. In contrast, our analysis does not rely on any assumptions regarding Gaussian distributions.

For the purposes of our theoretical results, we will need a data structure with stronger geometric properties than the simple kd-tree. Consider the basic kd-tree data structure described in the previous section. We define the *aspect ratio* of a cell to be the ratio of the length of its longest side to its shortest side. The *size* of a cell is the length of its longest side. The cells of a kd-tree may generally have arbitrarily high aspect ratio. For our analysis, we will assume that, rather than a kd-tree, the data points are stored in a structure called a *balanced box-decomposition tree* (or *BBD-tree*) for the point set [5]. The following two lemmas summarize the relevant properties of the BBD-tree. (See [5] for proofs.)

Lemma 1. *Given a set of n data points P in \mathbb{R}^d and a bounding hypercube C for the points, in $O(dn \log n)$ time it is possible to construct a BBD-tree representing a hierarchical decomposition of C into cells of complexity $O(d)$ such that:*

1. *The tree has $O(n)$ nodes and depth $O(\log n)$.*
2. *The cells have bounded aspect ratio and, with every $2d$ levels of descent in the tree, the sizes of the associated cells decrease by at least a factor of $1/2$.*

Lemma 2 (Packing Constraint). *Consider any set \mathcal{C} of cells of the BBD-tree with pairwise disjoint interiors, each of size at least s , that intersect a ball of radius r . The size of such a set is, at most, $(1 + \lceil \frac{4r}{s} \rceil)^d$.*

Our analysis is motivated by the observation that a great deal of the running time of Lloyd's algorithm is spent in the later stages of the algorithm when the center points are close to their final locations but the algorithm has not yet converged [25]. The analysis is based on the assumption that the data set

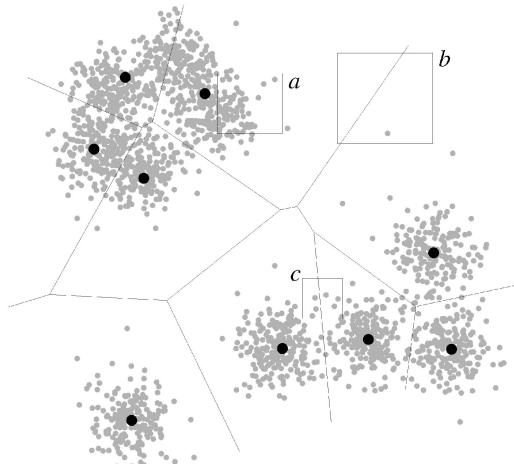


Fig. 3. Filtering algorithm analysis. Note that the density of points near the edges of the Voronoi diagram is relatively low.

can indeed be clustered into k natural clusters and that the current centers are located close to the true cluster centers. These are admittedly strong assumptions, but our experimental results will bear out the algorithm's efficiency even when these assumptions are not met.

We say that a node is *visited* if the Filter procedure is invoked on it. An internal node is *expanded* if its children are visited. A nonexpanded internal node or a leaf node is a *terminal node*. A nonleaf node is terminal if there is no center z that is closer to any part of the associated cell than the closest center z^* . Note that, if a nonleaf cell intersects the Voronoi diagram (that is, there is no center that is closest to every part of the cell), then it cannot be a terminal node. For example, in Fig. 3, the node with cell a is terminal because it lies entirely within the Voronoi cell of its nearest center. Cell b is terminal because it is a leaf. However, cell c is not terminal because it is not a leaf and it intersects the Voronoi diagram. Observe that, in this rather typical example, if the clusters are well-separated and the center points are close to the true cluster centers, then a relatively small fraction of the data set lies near the edges of the Voronoi diagram of the centers. The more well-separated the clusters, the smaller this fraction will be and, hence, fewer cells of the search structure will need to be visited by the algorithm. Our analysis formalizes this intuition.

We assume that the data points are generated independently from k different multivariate distributions in \mathbf{R}^d . Consider any one such distribution. Let $\mathbf{X} = (x_1, x_2, \dots, x_d)$ denote a random vector from this distribution. Let $\mu \in \mathbf{R}^d$ denote the mean point of this distribution and let Σ denote the $d \times d$ covariance matrix for the distribution [22]

$$\Sigma = E((\mathbf{X} - \mu)(\mathbf{X} - \mu)^T).$$

Observe that the diagonal elements of Σ are the variances of the random variables that are associated, respectively, with the individual coordinates. Let $\text{tr}(\Sigma)$ denote the *trace* of Σ , that is, the sum of its diagonal elements. We will measure the dispersion of the distribution by the variable $\sigma = \sqrt{\text{tr}(\Sigma)}$. This is a natural generalization of the notion of standard deviation for a univariate distribution.

We will characterize the degree of separation of the clusters by two parameters. Let $\mu^{(i)}$ and $\sigma^{(i)}$ denote, respectively, the mean and dispersion of the i th cluster distribution. Let

$$r_{\min} = \frac{1}{2} \min_{i \neq j} |\mu^{(i)} - \mu^{(j)}| \quad \text{and} \quad \sigma_{\max} = \max_i \sigma^{(i)},$$

where $|q - p|$ denotes the Euclidean distance between points q and p . The former quantity is half the minimum distance between any two cluster centers and the latter is the maximum dispersion. Intuitively, in well-separated clusters, r_{\min} is large relative to σ_{\max} . We define the *cluster separation* of the point distribution to be

$$\rho = \frac{r_{\min}}{\sigma_{\max}}.$$

This is similar to Dasgupta's notion of pairwise c -separated clusters, where $c = 2\rho$ [13]. It is also similar to the separation measure for cluster i of Coggins and Jain [12], defined to be $\min_{j \neq i} |\mu^{(i)} - \mu^{(j)}| / \sigma^{(i)}$.

We will show that, assuming that the candidate centers are relatively close to the cluster means, $\mu^{(i)}$, then, as ρ increases (i.e., as clusters are more well-separated) the algorithm's running time improves. Our proof makes use of the following straightforward generalization of Chebyshev's inequality (see [20]) to multivariate distributions. The proof proceeds by applying Chebyshev's inequality to each coordinate and summing the results over all d coordinates.

Lemma 3. *Let \mathbf{X} be a random vector in \mathbf{R}^d drawn from a distribution with mean μ and dispersion σ (the square root of the trace of the covariance matrix). Then, for all positive t ,*

$$\Pr(|\mathbf{X} - \mu| > t\sigma) \leq \frac{d}{t^2}.$$

For $\delta \geq 0$, we say that a set of candidates is δ -close with respect to a given set of clusters if, for each center $c^{(i)}$, there is an associated cluster mean $\mu^{(i)}$ within distance at most δr_{\min} and vice versa. Here is the main result of this section. Recall that a node is *visited* if the Filter procedure is invoked on it.

Theorem 1. *Consider a set of n points in \mathbf{R}^d drawn from a collection of cluster distributions with cluster separation $\rho = r_{\min}/\sigma_{\max}$, and consider a set of k candidate centers that are δ -close to the cluster means, for some $\delta < 1$. Then, for any $\epsilon, 0 < \epsilon < 1 - \delta$, and for some constant c , the expected number of nodes visited by the filtering algorithm is*

$$O\left(k \left(\frac{c\sqrt{d}}{\epsilon}\right)^d + 2^d k \log n + \frac{dn}{\rho^2(1 - \epsilon)^2}\right).$$

Before giving the proof, let us make a few observations about this result. The result provides an upper bound the number of nodes visited. The time needed to process each node is proportional to the number of candidates for the node, which is at most k and is typically much smaller. Thus, the total running time of the algorithm is larger by at most a factor of k . Also, the total running time includes an additive contribution of $O(dn \log n)$ time needed to build the initial BBD-tree.

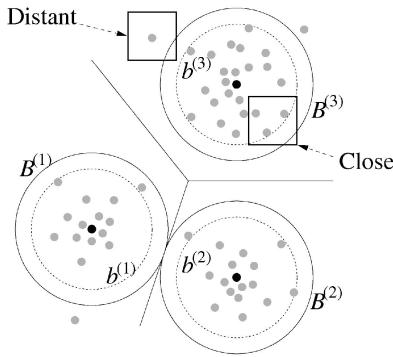


Fig. 4. Distant and close nodes.

We note that, for fixed d , ρ , and ϵ (bounded away from 0 and $1 - \delta$), the number of nodes visited as a function of n is $O(n)$ (since the last term in the analysis dominates). Thus, the overall running time is $O(kn)$, which seems to be no better than the brute-force algorithm. The important observation, however, is that, as cluster separation ρ increases, the $O(kn/\rho^2)$ running time decreases and the algorithm's efficiency increases rapidly. The actual expected number of cells visited may be much smaller for particular distributions.

Proof. Consider the i th cluster. Let $c^{(i)}$ denote a candidate center that is within distance δr_{\min} from its mean $\mu^{(i)}$. Let $\sigma^{(i)}$ denote its dispersion. Let $B^{(i)}$ denote a ball of radius r_{\min} centered at $\mu^{(i)}$. Observe that, because no cluster centers are closer than $2r_{\min}$, these balls have pairwise disjoint interiors. Let $b^{(i)}$ denote a ball of radius $r_{\min}(1 - \epsilon - \delta)$ centered at $c^{(i)}$. Because $c^{(i)}$ is δ -close to μ_i , $b^{(i)}$ is contained within a ball of radius $r_{\min}(1 - \epsilon)$ centered at $\mu^{(i)}$ and, hence, is contained within $B^{(i)}$. Moreover, the distance between the boundaries of $B^{(i)}$ and $b^{(i)}$ is at least ϵr_{\min} .

Consider a visited node and let C denote its associated cell in the BBD-tree. We consider two cases. First, suppose that, for some cluster i , $C \cap b^{(i)} \neq \emptyset$. We call C a *close node*. Otherwise, if the cell does not intersect any of the $b^{(i)}$ balls, it is a *distant node*. (See Fig. 4.) The intuition behind the proof is as follows: If clustering is strong, then we would expect only a few points to be far from the cluster centers and, hence, there are few distant nodes. If a node is close to its center and it is not too large, then there is only one candidate for the nearest neighbor and, hence, the node will be terminal. Because cells of the BBD tree have bounded aspect ratio, the number of large cells that can overlap a ball is bounded.

First, we bound the number of distant visited nodes. Consider the subtree of the BBD-tree induced by these nodes. If a node is distant, then its children are both distant, implying that this induced subtree is a full binary tree (that is, each nonleaf node has exactly two children). It is well-known that the total number of nonleaf nodes in a full binary tree is not greater than the number of leaves and, hence, it suffices to bound the number of leaves.

The data points associated with all the leaves of the induced subtree cannot exceed the number of data points that lie outside of $b^{(i)}$. As observed above, for any cluster i , a data point of this cluster that lies outside of $b^{(i)}$ is at distance from $\mu^{(i)}$ of at least

$$r_{\min}(1 - \epsilon) = \frac{r_{\min}}{\sigma^{(i)}}(1 - \epsilon)\sigma^{(i)} \geq \rho(1 - \epsilon)\sigma^{(i)}.$$

By Lemma 3, the probability of this occurring is at most $d/(\rho(1 - \epsilon))^2$. Thus, the expected number of such data points is at most $dn/(\rho(1 - \epsilon))^2$. It follows from standard results on binary tree that the number of distant of nodes is at most twice as large.

Next, we bound the number of close visited nodes. Recall a visited node is said to be *expanded* if the algorithm visits its children. Clearly, the number of close visited nodes is proportional to the number of close expanded nodes. For each cluster i , consider the leaves of the induced subtree consisting of close expanded nodes that intersect $b^{(i)}$. (The total number of close expanded nodes will be larger by a factor of k .) To bound the number of such nodes, we further classify them by the size of the associated cell. An expanded node v whose size is at least $4r_{\min}$ is *large* and, otherwise, it is *small*. We will show that the number of large expanded nodes is bounded by $O(2^d \log n)$ and the number of small expanded nodes is bounded by $O((c\sqrt{d}/\epsilon)^d)$, for some constant c .

We first show the bound on the number of large expanded nodes. In the descent through the BBD-tree, the sizes of the nodes decrease monotonically. Consider the set of all expanded nodes of size greater than $4r_{\min}$. These nodes induce a subtree in the BBD-tree. Let L denote the leaves of this tree. The cells associated with the elements of L have pairwise disjoint interiors and they intersect $b^{(i)}$ and, hence, they intersect $B^{(i)}$. It follows from Lemma 2 (applied to $B^{(i)}$ and the cells associated with L) that there are at most $(1 + \lceil 4r_{\min}/(4r_{\min}) \rceil)^d = 2^d$ such cells. By Lemma 1, the depth of the tree is $O(\log n)$ and, hence, the total number of expanded large nodes is $O(2^d \log n)$, as desired.

Finally, we consider the small expanded nodes. We assert that the diameter of the cell corresponding to any such node is at least ϵr_{\min} . If not, then this cell would lie entirely within a ball of radius $r_{\min}(1 - \delta)$ of $c^{(i)}$. Since the cell was expanded, we know that there must be a point in this cell that is closer to some other center c_j . This implies that the distance between $c^{(i)}$ and $c^{(j)}$ is less than $2r_{\min}(1 - \delta)$. Since the candidates are δ -close, it follows that there are two cluster means $\mu^{(i)}$ and $\mu^{(j)}$ that are closer than $2r_{\min}(1 - \delta) + 2\delta r_{\min} = 2r_{\min}$. However, this would contradict the definition of r_{\min} . A cell in dimension d of diameter x has longest side length of at least x/\sqrt{d} . Thus, the size of each such cell is at least $\epsilon r_{\min}/\sqrt{d}$. By Lemma 2, it follows that the number of such cells that overlap $B^{(i)}$ is at most

$$\left(\frac{c\sqrt{d}}{\epsilon}\right)^d.$$

Applying a similar analysis used by Arya and Mount [4] for approximate range searching, the number of expanded nodes is asymptotically the same. This completes the proof. \square

4 EMPIRICAL ANALYSIS

To establish the practical efficiency of the filtering algorithm, we implemented it and tested its performance on a number of

data sets. These included both synthetically generated data and data used in real applications. The algorithm was implemented in C++ using the g++ compiler and was run on a Sun Ultra 5 running Solaris 2.6. The data structure implemented was a kd-tree that was constructed from the ANN library [43] using the sliding midpoint rule [39]. This decomposition method was chosen because our studies have shown that it performs better than the standard kd-tree decomposition rule for clustered data sets. Essentially, we performed two different types of experiments. The first type compared running times of the filtering algorithm against two different implementations of Lloyd's algorithm. The second type compared cluster distortions obtained for the filtering algorithm with those obtained for the BIRCH clustering scheme [50].

For the running time experiments, we considered two other algorithms. Recall that the essential task is to compute the closest center to each data point. The first algorithm compared against was a simple *brute-force* algorithm, which computes the distance from every data point to every center. The second algorithm, called *kd-center*, operates by building a kd-tree with respect to the center points and then uses the kd-tree to compute the nearest neighbor for each data point. The kd-tree is rebuilt at the start of each stage of Lloyd's algorithm. We compared these two methods against the filtering algorithm by performing two sets of experiments, one involving synthetic data and the other using data derived from applications in image segmentation and compression.

We used the following experimental structure for the above experiments: For consistency, we ran all three algorithms on the same data set and the same initial placement of centers. Because the running time of the algorithm depends heavily on the number of stages, we report the average running time per stage by computing the total running time and then dividing by the number of stages. In the case of the filtering algorithm, we distinguished between two cases according to whether or not the preprocessing time was taken into consideration. The reason for excluding preprocessing time is that, when the number of stages of Lloyd's algorithm is very small, the effect of the preprocessing time is amortized over a smaller number of stages and this introduces a bias.

We measured running time in two different ways. We measured both the CPU time (using the standard `clock()` function) and a second quantity called the number of *node-candidate pairs*. The latter quantity is a machine-independent statistic of the algorithm's complexity. Intuitively, it measures the number of interactions between a node of the kd-tree (or data point in the case of brute force) and a candidate center. For the brute-force algorithm, this quantity is always kn . For the kd-center algorithm, for each data point, we count the number of nodes that were accessed in the kd-tree of the centers for computing its nearest center and sum this over all data points. For the filtering algorithm, we computed a sum of the number of candidates associated with every visited node of the tree. In particular, for each call to $\text{Filter}(u, Z)$, the cardinality of the set Z of candidate centers is accumulated. The one-time cost of building the kd-tree is not included in this measure.

Note that the three algorithms are functionally equivalent if points are in general position. Thus, they all produce the same final result and, so, there is no issue regarding the quality of the final output. The primary issue of interest is the efficiency of the computation.

4.1 Synthetic Data

We ran three experiments to determine the variations in running time as a function of cluster separation, data set size, and dimension. The first experiment tested the validity of Theorem 1. We generated $n = 10,000$ data points in \mathbf{R}^3 . These points were distributed evenly among 50 clusters as follows: The 50 cluster centers were sampled from a uniform distribution over the hypercube $[-1, 1]^d$. A Gaussian distribution was then generated around each center, where each coordinate was generated independently from a univariate Gaussian with a given standard deviation. The standard deviation varied from 0.01 (very well-separated) up to 0.7 (virtually unclustered). Because the same distribution was used for cluster centers throughout, the expected distances between cluster centers remained constant. Thus, the expected value of the cluster separation parameter ρ varied inversely with the standard deviation. The initial centers were chosen by taking a random sample of data points.

For each standard deviation, we ran each of the algorithms (brute-force, kd-center, and filter) three times. For each of the three runs, a new set of initial center points was generated and all three algorithms were run using the same data and initial center points. The algorithm ran for a maximum of 30 stages or until convergence.

The average CPU times per stage, for all three methods, are shown for $k = 50$ in Fig. 5a and for $k = 20$ in Fig. 6a. The results of these experiments show that the filtering algorithm runs significantly faster than the other two algorithms. Ignoring the bias introduced by preprocessing, its running time improves when the clusters are more well-separated (for smaller standard deviations). The improvement in CPU time predicted by Theorem 1 is not really evident for very small standard deviations because initialization and preprocessing costs dominate. However, this improvement is indicated in Figs. 5b and 6b, which plot the (initialization independent) numbers of node-candidate pairs. The numbers of node candidate pairs for the other two methods are not shown, but varied from 4 to 10 times greater than those of the filtering algorithm.

Our theoretical analysis does not predict that the filtering algorithm will be particularly efficient for unclustered data sets, but it does not preclude this possibility. Nonetheless, we observe in this experiment that the filtering algorithm ran significantly faster than the brute-force and kd-center algorithms, even for large standard deviations. This can be attributed, in part, to the fact that the filtering algorithm simply does a better job in exploiting economies of scale by storing the much larger set of data points in a kd-tree (rather than center points as kd-center does).

The objective of the second experiment was to study the effects of data size on the running time. We generated data sets of points whose size varied from $n = 1,000$ to $n = 20,000$ and where each data set consisted of 50 Gaussian clusters. The standard deviation was fixed at 0.10 and the algorithms were run with $k = 50$. Again, the searches were terminated when stability was achieved or after 30 stages. As before, we ran each case three times with different starting centers and averaged the results. The CPU times per stage and number of

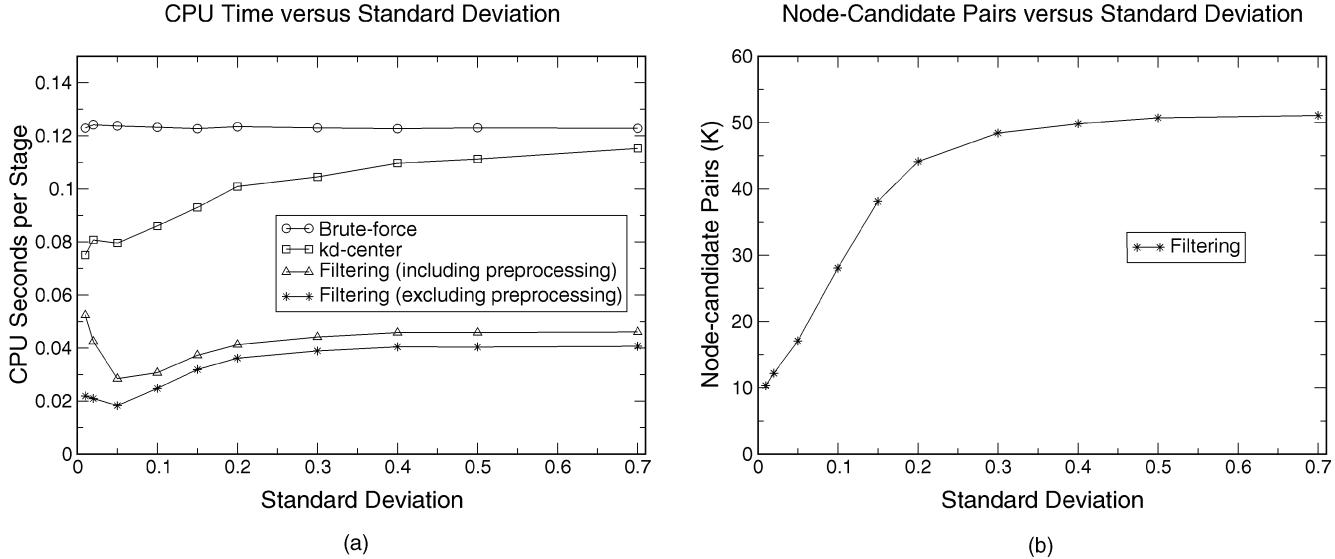


Fig. 5. Average CPU times and node-candidate pairs per stage versus cluster standard deviation for $n = 10,000$, $k = 50$.

node-candidate pairs for all three methods are shown in Fig. 7. The results of this experiment show that, for fixed k and large n , all three algorithms have running times that vary linearly with n , with the filtering algorithm doing the best.

The third experiment was designed to test the effect of the dimension on running times of the three k -means algorithms. Theorem 1 predicts that the filtering algorithm's running time increases exponentially with dimension, whereas that of the brute-force algorithm (whose running time is $O(dkn)$) increases only linearly with dimension d . Thus, as the dimension increases, we expect that the filtering algorithm's speed advantage would tend to diminish. This exponential dependence on dimension seems to be characteristic of many algorithms that are based on kd-trees and many variants, such as R-trees.

We repeated an experimental design similar to the one used by Pelleg and Moore [45]. For each dimension d , we created a random data set with 20,000 points. The points were sampled from a clustered Gaussian distribution with 72 clusters and a standard deviation of 0.05 along each coordinate. (The standard deviation is twice that used by Pelleg and Moore because they used a unit hypercube and our hypercube has side length 2.) The three k -means algorithms were run with $k = 40$ centers. In Fig. 8, we plot the average times taken per stage for the three algorithms. We found that, for this setup, the filtering algorithm outperforms the brute-force and kd-center algorithms for dimensions ranging up to the mid 20s. These results confirm the general trend reported in [45], but the filtering algorithm's performance in moderate dimensions 10 to 20 is considerably better.

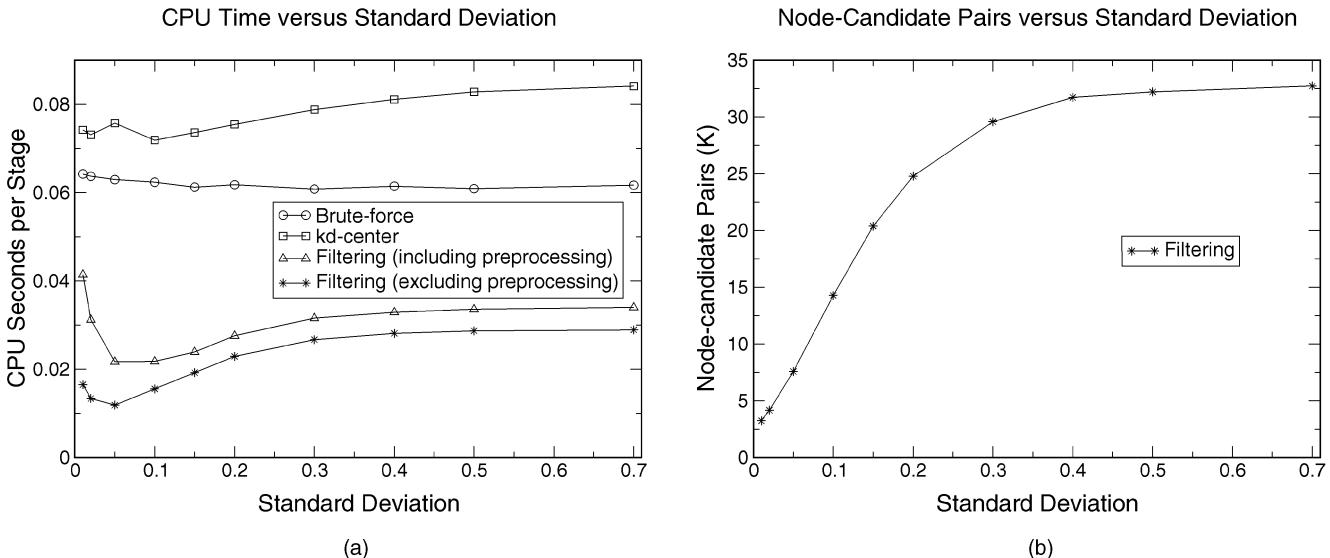


Fig. 6. Average CPU times and node-candidate pairs per stage versus cluster standard deviation for $n = 10,000$, $k = 20$.

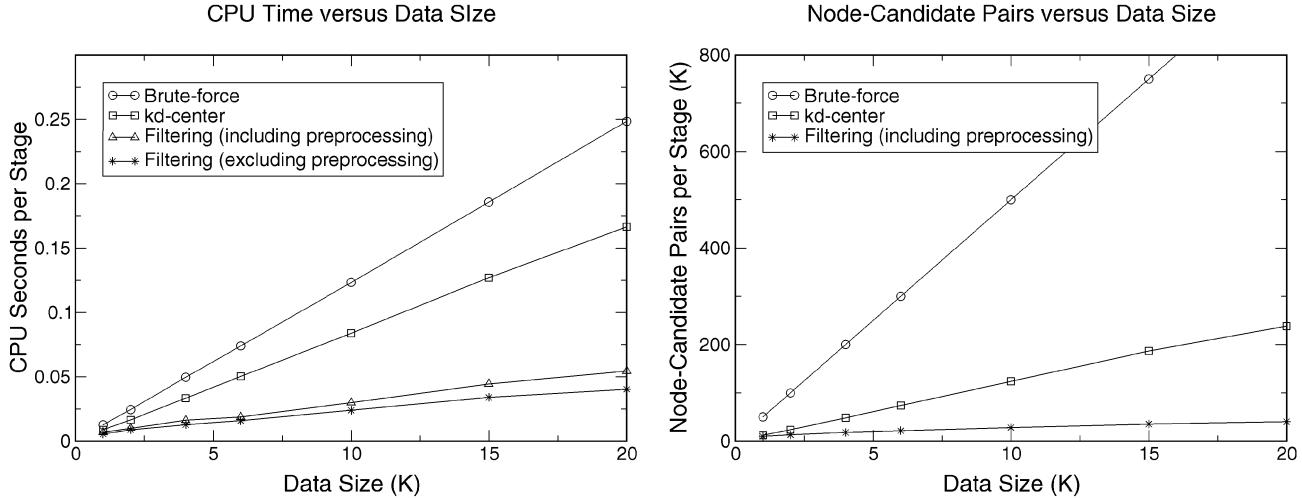


Fig. 7. Running times and node-candidate pairs versus data size for $k = 50$, $\sigma = 0.10$.

4.2 Real Data

To understand the relative efficiency of this algorithm under more practical circumstances, we ran a number of experiments on data sets derived from actual applications in image processing, compression, and segmentation. Each experiment involved solving a k -means problem on a set of points in \mathbb{R}^d for various d s. In each case, we applied all three algorithms: brute force (Brute), kd-center (KDcenter), and filtering (Filter). In the filtering algorithm, we included the preprocessing time in the average. In each case, we computed the average CPU time per stage in seconds (T-) and the average number of node-candidate pairs (NC-). This was repeated for values of k chosen from $\{8, 64, 256\}$. The results are shown in Table 1.

The experiments are grouped into three general categories. The first involves a color quantization application. A color image is given and a number of pixels are sampled from the image (10,000 for this experiment). Each pixel is represented as a triple consisting of red, green, and blue components, each in the range $[0, 255]$ and, hence, is a point in \mathbb{R}^3 . The input images were chosen from a number of

standard images for this application. The images are shown in Fig. 9 and the running time results are given in the upper half of Table 1, opposite the corresponding image names "balls," "kiss," ..., "ball1_s."

The next experiment involved a vector quantization application for data compression. Here, the images are gray-scale images with pixel values in the range $[0, 255]$. Each 2×2 subarray of pixels is selected and mapped to a 4-element vector and the k -means algorithm is run on the resulting set of vectors. The images are shown in Fig. 10 and the results are given in Table 1, opposite the names "couple," "crowd," ..., "woman2."

The final experiment involved image segmentation. A 512×512 Landsat image of Israel consisting of four spectral bands was used. The resulting 4-element vectors in the range $[0, 255]$ were presented to the algorithms. One of the image bands is shown in Fig. 11 and the results are provided in Table 1, opposite the name "Israel."

An inspection of the results reveals that the filtering algorithm significantly outperformed the other two algorithms in all cases. Plots of the underlying point distributions showed that most of these data sets were really not well-clustered. Thus, the filtering algorithm is quite efficient, even when the conditions of Theorem 1 are not satisfied.

4.3 Comparison with BIRCH

A comparison between our filtering algorithm and the BIRCH clustering scheme [50] is presented in Table 2. The table shows the distortions produced by both algorithms. The column "BIRCH" reports the distortions obtained with the BIRCH software. The column labeled "Filter" provides the distortions obtained due to the k -means filtering algorithm (with centers initially sampled at random from the data points). Alternatively, one can use the centers obtained by BIRCH as initial centers for our filtering algorithm. This is shown in the column labeled "Combined" of the table. As can be seen, the distortions are always better and considerably better in some of the cases. (This is because k -means cannot increase distortions.) Indeed, as was mentioned in Section 1, k -means may be run to improve the distortion of any clustering algorithm.

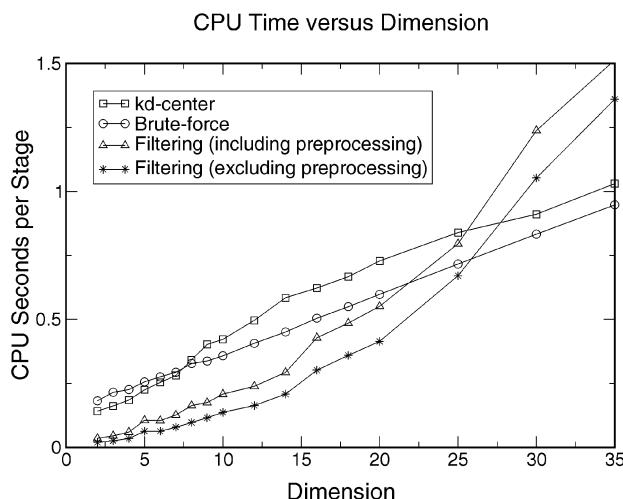


Fig. 8. Average CPU times per stage versus dimension for $n = 20,000$, $\sigma = 0.05$, and $k = 40$.

TABLE 1
Running Times for Various Test Inputs

Image	Dim	Size	k	T-Brute	T-KDcenter	T-Filter	NC-Brute	NC-KDcenter	NC-Filter
balls	3	10000	8	0.0542857	0.0804762	0.0157143	80	68.13	3.678
			64	0.20875	0.1515	0.01875	640	160.9	17.65
			256	0.742424	0.17875	0.0375758	2560	180	49.77
kiss	3	10000	8	0.053	0.08225	0.0165	80	71.92	10.07
			64	0.209	0.14875	0.0435	640	170.4	41.76
			256	0.7455	0.218	0.089	2560	243.9	111
Lena1	3	10000	8	0.0538889	0.0797222	0.0163889	80	65.18	9.767
			64	0.20925	0.14575	0.046	640	164.7	43.72
			256	0.75025	0.218	0.0915	2560	236.8	118.2
ball1_h	3	10000	8	0.0535	0.0755	0.0095	80	57.27	3.859
			64	0.207	0.119118	0.0245	640	117.8	20.98
			256	0.749231	0.16575	0.0584615	2560	167.2	69.42
ball1_3	3	10000	8	0.0535	0.07625	0.0095	80	58.88	4.073
			64	0.208571	0.11641	0.0251429	640	115.9	20.69
			256	0.748846	0.166364	0.0553846	2560	167.8	65.83
ball1_5	3	10000	8	0.0541667	0.08	0.0225	80	62.72	2.712
			64	0.214688	0.125278	0.029375	640	122.4	23.16
			256	0.74775	0.178	0.059	2560	175.5	71.89
ball1_p	3	10000	8	0.0535714	0.0778571	0.0207143	80	64.37	3.12
			64	0.2095	0.122	0.0285	640	122.2	23.02
			256	0.743043	0.17	0.0608333	2560	174.2	72.93
ball1_s	3	10000	8	0.053	0.0771429	0.015	80	57.95	3.497
			64	0.20875	0.123939	0.0275	640	123.5	24.04
			256	0.745	0.17225	0.059	2560	176	74.3
couple	4	65536	8	0.37475	0.5575	0.15825	524.3	482.3	74.6
			64	1.616	1.21925	0.3535	4194	1452	285.8
			256	5.501	1.732	0.658	1.678e+04	2375	739.6
crowd	4	65536	8	0.36875	0.57375	0.146	524.3	497.7	62.35
			64	1.60975	1.304	0.32175	4194	1512	244.9
			256	5.49975	1.716	0.535	1.678e+04	2347	567.3
lax	4	65536	8	0.36675	0.59225	0.18475	524.3	558.4	99.04
			64	1.6085	1.3725	0.42325	4194	1772	359.7
			256	5.495	1.86175	0.7055	1.678e+04	2776	790.6
Lena2	4	65536	8	0.373	0.55475	0.123	524.3	471.5	50.83
			64	1.615	1.19375	0.34475	4194	1339	266.9
			256	5.50475	1.649	0.618	1.678e+04	2200	677.1
man	4	65536	8	0.3685	0.5505	0.13925	524.3	465.9	63.16
			64	1.609	1.20275	0.35375	4194	1392	278.1
			256	5.49975	1.66125	0.60675	1.678e+04	2275	665.4
woman1	4	65536	8	0.370937	0.563125	0.176875	524.3	483.4	73.65
			64	1.612	1.28275	0.3805	4194	1500	304.2
			256	5.523	1.7945	0.64325	1.678e+04	2450	706.8
woman2	4	65536	8	0.37175	0.544	0.1125	524.3	433.3	35.46
			64	1.615	1.20075	0.30775	4194	1277	214.2
			256	5.51675	1.64675	0.519	1.678e+04	2133	531.4
Israel	4	262144	8	1.6955	3.06775	0.7295	2097	2079	140.4
			64	5.82975	4.82825	1.55225	1.678e+04	5336	558.5
			256	19.975	6.659	2.53775	6.711e+04	8305	1148

5 CONCLUDING REMARKS

We have presented an efficient implementation of Lloyd's k -means clustering algorithm, called the filtering algorithm. The algorithm is easy to implement and only requires that a

kd-tree be built once for the given data points. Efficiency is achieved because the data points do not vary throughout the computation and, hence, this data structure does not need to be recomputed at each stage. Since there are typically many more data points than "query" points (i.e., centers), the

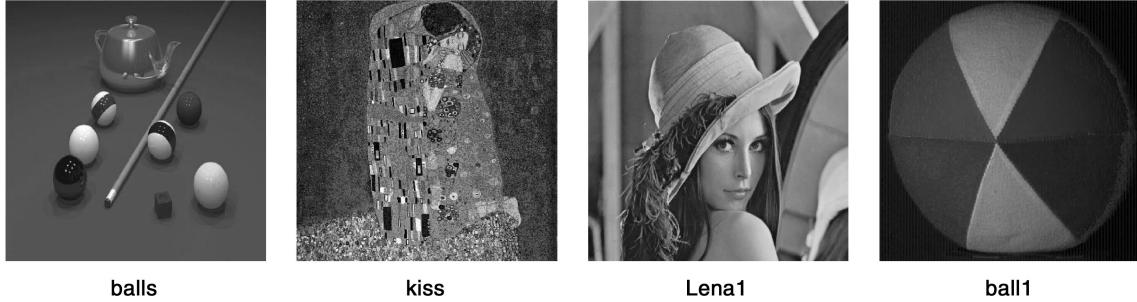


Fig. 9. Sample of images in the color quantization data set. Each pixel in the RGB color images is represented by a triple with values in the range [0, 255].

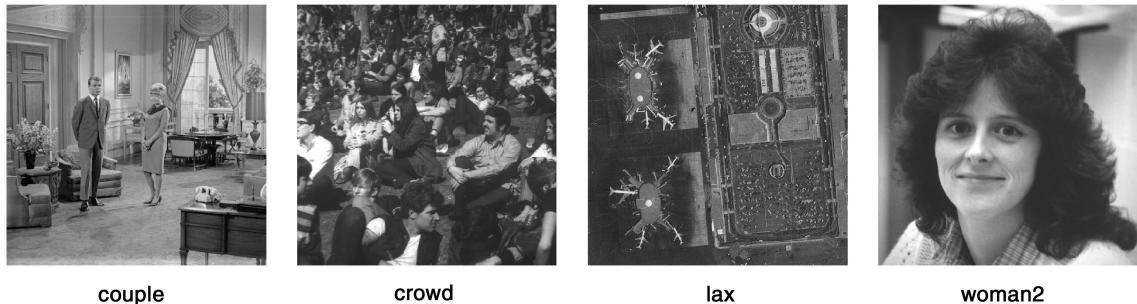


Fig. 10. Sample of images in the image compression dataset. Each pixel in these gray-scale images is represented by 8 bits. 2×2 nonoverlapping blocks were used to create 4-element vectors.

relative advantage provided by preprocessing in the above manner is greater. Our algorithm differs from existing approaches only in how nearest centers are computed, so it could be applied to the many variants of Lloyd's algorithm.

The algorithm has been implemented and the source code is available on request. We have demonstrated the practical efficiency of this algorithm both theoretically, through a data sensitive analysis, and empirically, through experiments on both synthetically generated and real data sets. The data sensitive analysis shows that the more well-separated the clusters, the faster the algorithm runs. This is subject to the assumption that the center points are indeed close to the cluster centers. Although this assumption is rather strong, our empirical analysis on synthetic data

indicates that the algorithm's running time does improve dramatically as cluster separation increases. These results for both synthetic and real data sets indicate that the filtering algorithm is significantly more efficient than the other two methods that were tested. Further, they show that the algorithm scales well up to moderately high dimensions (from 10 to 20). Its performance in terms of distortions is competitive with that of the well-known BIRCH software and, by running k -means as a postprocessing step to BIRCH, it is possible to improve distortions significantly.

A natural question is whether the filtering algorithm can be improved. The most obvious source of inefficiency in the algorithm is that it passes no information from one stage to the next. Presumably, in the later stages of Lloyd's algorithm, as the centers are converging to their final positions, one would expect that the vast majority of the data points have the same closest center from one stage to the next. A good algorithm should exploit this coherence to improve the running time. A kinetic method along these lines was proposed in [31], but this algorithm is quite complex and does not provide significantly faster running time in practice. The development of a simple and practical algorithm which combines the best elements of the kinetic and filtering approaches would make a significant contribution.

ACKNOWLEDGMENTS

The authors would like to thank Azriel Rosenfeld for his comments and Hao Li for his help in running the BIRCH experiments. They also are grateful to the anonymous referees for their many valuable suggestions. This research was funded in part by the US National Science Foundation under Grant CCR-0098151 and by the US Army Research Laboratory and the US Department of Defense under



Fig. 11. Images in satellite image data set. One of four spectral bands of a Landsat image over Israel is shown. The image is 512×512 and each pixel is represented by 8 bits.

TABLE 2
Average Distortions for the Filtering Algorithm, BIRCH, and the Two Combined

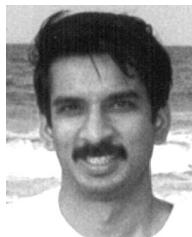
Image	Dim	BIRCH	Filter	Combined	Image	Dim	BIRCH	Filter	Combined
balls	8	697.10	623.20	434.70	couple	8	766.70	594.00	631.64
	64	53.22	116.50	38.81		64	223.80	182.10	178.44
	256	28.54	28.74	11.04		256	131.20	94.21	89.71
kiss	8	944.50	718.60	687.42	crowd	8	539.70	464.60	458.75
	64	171.20	152.40	142.49		64	196.10	159.10	137.03
	256	75.33	58.36	53.50		256	106.20	89.02	65.66
Lena	8	502.60	447.20	406.68	lax	8	935.40	840.50	848.86
	64	94.78	78.83	75.80		64	426.20	314.10	305.41
	256	34.76	29.80	27.84		256	288.30	169.00	156.74
ball1.h	8	531.00	716.70	501.59	Lena4	8	367.90	368.10	340.12
	64	59.75	61.32	47.21		64	136.70	101.10	103.70
	256	32.00	21.05	15.61		256	80.89	51.79	52.87
ball1..3	8	440.20	663.20	423.89	man	8	433.20	455.10	428.93
	64	37.39	39.30	31.62		64	199.60	152.60	153.08
	256	17.85	11.86	9.43		256	148.90	79.53	77.18
ball1..5	8	490.40	421.50	469.02	woman1	8	549.10	509.40	510.47
	64	58.21	56.98	46.12		64	216.80	169.80	160.63
	256	27.53	18.20	14.98		256	147.90	88.54	85.73
ball1..p	8	574.60	724.20	525.84	woman2	8	259.80	252.80	234.11
	64	79.75	62.92	55.15		64	50.71	46.67	41.90
	256	47.72	21.99	18.64		256	31.00	24.14	20.90
ball1..s	8	570.20	535.00	541.79	Israel	8	421.9	397.31	389.06
	64	75.58	70.61	59.74		64	93.34	84.09	82.72
	256	42.22	25.05	19.69		256	44.20	34.90	34.29

Contract MDA9049-6C-1250. A preliminary version of this paper appeared in the *Proceedings of the 16th Annual ACM Symposium on Computational Geometry*, pp. 100-109, June 2000.

REFERENCES

- [1] P.K. Agarwal and C.M. Procopiuc, "Exact and Approximation Algorithms for Clustering," *Proc. Ninth Ann. ACM-SIAM Symp. Discrete Algorithms*, pp. 658-667, Jan. 1998.
- [2] K. Alsabti, S. Ranka, and V. Singh, "An Efficient k -means Clustering Algorithm," *Proc. First Workshop High Performance Data Mining*, Mar. 1998.
- [3] S. Arora, P. Raghavan, and S. Rao, "Approximation Schemes for Euclidean k -median and Related Problems," *Proc. 30th Ann. ACM Symp. Theory of Computing*, pp. 106-113, May 1998.
- [4] S. Arya and D. M. Mount, "Approximate Range Searching," *Computational Geometry: Theory and Applications*, vol. 17, pp. 135-163, 2000.
- [5] S. Arya, D.M. Mount, N.S. Netanyahu, R. Silverman, and A.Y. Wu, "An Optimal Algorithm for Approximate Nearest Neighbor Searching," *J. ACM*, vol. 45, pp. 891-923, 1998.
- [6] G.H. Ball and D.J. Hall, "Some Fundamental Concepts and Synthesis Procedures for Pattern Recognition Preprocessors," *Proc. Int'l Conf. Microwaves, Circuit Theory, and Information Theory*, Sept. 1964.
- [7] J.L. Bentley, "Multidimensional Binary Search Trees Used for Associative Searching," *Comm. ACM*, vol. 18, pp. 509-517, 1975.
- [8] L. Bottou and Y. Bengio, "Convergence Properties of the k -means Algorithms," *Advances in Neural Information Processing Systems 7*, G. Tesauro and D. Touretzky, eds., pp. 585-592. MIT Press, 1995.
- [9] P.S. Bradley and U. Fayyad, "Refining Initial Points for K-means Clustering," *Proc. 15th Int'l Conf. Machine Learning*, pp. 91-99, 1998.
- [10] P.S. Bradley, U. Fayyad, and C. Reina, "Scaling Clustering Algorithms to Large Databases," *Proc. Fourth Int'l Conf. Knowledge Discovery and Data Mining*, pp. 9-15, 1998.
- [11] V. Capoyleas, G. Rote, and G. Woeginger, "Geometric Clusterings," *J. Algorithms*, vol. 12, pp. 341-356, 1991.
- [12] J.M. Coggins and A.K. Jain, "A Spatial Filtering Approach to Texture Analysis," *Pattern Recognition Letters*, vol. 3, pp. 195-203, 1985.
- [13] S. Dasgupta, "Learning Mixtures of Gaussians," *Proc. 40th IEEE Symp. Foundations of Computer Science*, pp. 634-644, Oct. 1999.
- [14] S. Dasgupta and L.J. Shulman, "A Two-Round Variant of EM for Gaussian Mixtures," *Proc. 16th Conf. Uncertainty in Artificial Intelligence (UAI-2000)*, pp. 152-159, June 2000.
- [15] Q. Du, V. Faber, and M. Gunzburger, "Centroidal Voronoi Tesselations: Applications and Algorithms," *SIAM Rev.*, vol. 41, pp. 637-676, 1999.
- [16] R.O. Duda and P.E. Hart, *Pattern Classification and Scene Analysis*. New York: John Wiley & Sons, 1973.
- [17] M. Ester, H. Kriegel, and X. Xu, "A Database Interface for Clustering in Large Spatial Databases," *Proc. First Int'l Conf. Knowledge Discovery and Data Mining (KDD-95)*, pp. 94-99, 1995.
- [18] V. Faber, "Clustering and the Continuous k -means Algorithm," *Los Alamos Science*, vol. 22, pp. 138-144, 1994.
- [19] U.M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, *Advances in Knowledge Discovery and Data Mining*. AAAI/MIT Press, 1996.
- [20] W. Feller, *An Introduction to Probability Theory and Its Applications*, third ed. New York: John Wiley & Sons, 1968.
- [21] E. Forgy, "Cluster Analysis of Multivariate Data: Efficiency vs. Interpretability of Classification," *Biometrics*, vol. 21, p. 768, 1965.
- [22] K. Fukunaga, *Introduction to Statistical Pattern Recognition*. Boston: Academic Press, 1990.
- [23] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York: W.H. Freeman, 1979.
- [24] A. Gersho and R.M. Gray, *Vector Quantization and Signal Compression*. Boston: Kluwer Academic, 1992.
- [25] M. Inaba, private communication, 1997.
- [26] M. Inaba, H. Imai, and N. Katoh, "Experimental Results of a Randomized Clustering Algorithm," *Proc. 12th Ann. ACM Symp. Computational Geometry*, pp. C1-C2, May 1996.

- [27] M. Inaba, N. Katoh, and H. Imai, "Applications of Weighted Voronoi Diagrams and Randomization to Variance-Based k -clustering," *Proc. 10th Ann. ACM Symp. Computational Geometry*, pp. 332-339, June 1994.
- [28] A.K. Jain and R.C. Dubes, *Algorithms for Clustering Data*. Englewood Cliffs, N.J.: Prentice Hall, 1988.
- [29] A.K. Jain, P.W. Duin, and J. Mao, "Statistical Pattern Recognition: A Review," *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. 22, no. 1, pp. 4-37, Jan. 2000.
- [30] A.K. Jain, M.N. Murty, and P.J. Flynn, "Data Clustering: A Review," *ACM Computing Surveys*, vol. 31, no. 3, pp. 264-323, 1999.
- [31] T. Kanungo, D.M. Mount, N.S. Netanyahu, C. Piatko, R. Silverman, and A.Y. Wu, "Computing Nearest Neighbors for Moving Points and Applications to Clustering," *Proc. 10th Ann. ACM-SIAM Symp. Discrete Algorithms*, pp. S931-S932, Jan. 1999.
- [32] T. Kanungo, D.M. Mount, N.S. Netanyahu, C. Piatko, R. Silverman, and A.Y. Wu, "The Analysis of a Simple k -means Clustering Algorithm," Technical Report CAR-TR-937, Center for Automation Research, Univ. of Maryland, College Park, Jan. 2000.
- [33] T. Kanungo, D.M. Mount, N.S. Netanyahu, C. Piatko, R. Silverman, and A.Y. Wu, "The Analysis of a Simple k -means Clustering Algorithm," *Proc. 16th Ann. ACM Symp. Computational Geometry*, pp. 100-109, June 2000.
- [34] L. Kaufman and P.J. Rousseeuw, *Finding Groups in Data: An Introduction to Cluster Analysis*. New York: John Wiley & Sons, 1990.
- [35] T. Kohonen, *Self-Organization and Associative Memory*, third ed. New York: Springer-Verlag, 1989.
- [36] S. Kolliopoulos and S. Rao, "A Nearly Linear-Time Approximation Scheme for the Euclidean k -median Problem," *Proc. Seventh Ann. European Symp. Algorithms*, J. Nešetřil, ed., pp. 362-371, July 1999.
- [37] S. P. Lloyd, "Least Squares Quantization in PCM," *IEEE Trans. Information Theory*, vol. 28, 129-137, 1982.
- [38] J. MacQueen, "Some Methods for Classification and Analysis of Multivariate Observations," *Proc. Fifth Berkeley Symp. Math. Statistics and Probability*, vol. 1, pp. 281-296, 1967.
- [39] S. Maneewongvatana and D.M. Mount, "Analysis of Approximate Nearest Neighbor Searching with Clustered Point Sets," *Proc. Workshop Algorithm Eng. and Experiments (ALENEX '99)*, Jan. 1999. Available from: <http://ftp.cs.umd.edu/pub/faculty/mount/Papers/dimacs99.ps.gz>.
- [40] O.L. Mangasarian, "Mathematical Programming in Data Mining," *Data Mining and Knowledge Discovery*, vol. 1, pp. 183-201, 1997.
- [41] J. Matousek, "On Approximate Geometric k -clustering," *Discrete and Computational Geometry*, vol. 24, pp. 61-84, 2000.
- [42] A. Moore, "Very Fast EM-Based Mixture Model Clustering Using Multiresolution kd-Trees," *Proc. Conf. Neural Information Processing Systems*, 1998.
- [43] D.M. Mount and S. Arya, "ANN: A Library for Approximate Nearest Neighbor Searching," *Proc. Center for Geometric Computing Second Ann. Fall Workshop Computational Geometry*, 1997. (available from <http://www.cs.umd.edu/~mount/ANN/>)
- [44] R.T. Ng and J. Han, "Efficient and Effective Clustering Methods for Spatial Data Mining," *Proc. 20th Int'l Conf. Very Large Databases*, pp. 144-155, Sept. 1994.
- [45] D. Pelleg and A. Moore, "Accelerating Exact k -means Algorithms with Geometric Reasoning," *Proc. ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining*, pp. 277-281, Aug. 1999.
- [46] D. Pelleg and A. Moore, " x -means: Extending k -means with Efficient Estimation of the Number of Clusters," *Proc. 17th Int'l Conf. Machine Learning*, July 2000.
- [47] D. Pollard, "A Central Limit Theorem for k -means Clustering," *Annals of Probability*, vol. 10, pp. 919-926, 1982.
- [48] F.P. Preparata, M.I. Shamos, *Computational Geometry: An Introduction*, third ed. Springer-Verlag, 1990.
- [49] S.Z. Selim and M.A. Ismail, "K-means-type Algorithms: A Generalized Convergence Theorem and Characterization of Local Optimality," *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. 6, pp. 81-87, 1984.
- [50] T. Zhang, R. Ramakrishnan, and M. Livny, "BIRCH: A New Data Clustering Algorithm and Its Applications," *Data Mining and Knowledge Discovery*, vol. 1, no. 2, pp. 141-182, 1997. (software available from www.cs.wisc.edu/~vganti/birchcode).



Tapas Kanungo received the PhD degree from the University of Washington, Seattle. Currently, he is a research staff member at the IBM Almaden Research Center, San Jose, California. Prior to working at IBM, he served as a codirector of the Language and Media Processing Lab at the University of Maryland, College Park. His current interests are in information extraction and retrieval and document analysis. He is senior member of the IEEE and the IEEE Computer Society.



David M. Mount received the PhD degree from Purdue University in computer science in 1983. He is a professor in the Department of Computer Science at the University of Maryland with a joint appointment at the Institute for Advanced Computer Studies. His primary research interests include the design, analysis, and implementation of data structures and algorithms for geometric problems, particularly problems with applications in image processing, pattern recognition, information retrieval, and computer graphics. He is an associate editor for *Pattern Recognition*. He is a member of the IEEE.



Nathan S. Netanyahu received BSc and MSc degrees in electrical engineering from the Technion, Israel Institute of Technology, and the MSc and PhD degrees in computer science from the University of Maryland, College Park. He is currently a senior lecturer in the Department of Mathematics and Computer Science at Bar-Ilan University and is also affiliated with the Center for Automation Research, University of Maryland, College Park. Dr. Netanyahu's main research interests are in the areas of algorithm design and analysis, computational geometry, image processing, pattern recognition, remote sensing, and robust statistical estimation. He currently serves as an associate editor for *Pattern Recognition*. He is a member of the IEEE.



Christine D. Piatko received the BA degree in computer science and mathematics from New York University in 1986 and the MS and PhD degrees from Cornell University in 1989 and 1993, respectively. She is currently a computer science researcher at The Johns Hopkins University Applied Physics Laboratory. Her research interests include computational geometry and information retrieval.



Ruth Silverman received the PhD degree in mathematics from the University of Washington in 1970. She is a retired professor from the Department of Computer Science at the University of the District of Columbia and she is a visiting professor at the Center for Automation Research at the University of Maryland, College Park.



Angela Y. Wu received the PhD degree in computer science from the University of Maryland at College Park in 1978. She is a professor in the Department of Computer Science and Information Systems, American University, Washington D.C., where she has taught since 1980. She is a senior member of the IEEE.