Published in final edited form as:
Proc Int Jt Conf Neural Netw. 2012 February 8; 43(6): 2351–2358. doi:10.1016/j.patcog.2010.01.003.

A Fast Exact *k*-Nearest Neighbors Algorithm for High Dimensional Search Using *k*-Means Clustering and Triangle Inequality

Xueyi Wang
X. Wang is with the Department of Mathematics and Computer Science, Northwest Nazarene University, Nampa, ID 83642 USA

Abstract

The *k*-nearest neighbors (*k*-NN) algorithm is a widely used machine learning method that finds nearest neighbors of a test object in a feature space. We present a new exact *k*-NN algorithm called *k*M*k*NN (*k*-Means for *k*-Nearest Neighbors) that uses the *k*-means clustering and the triangle inequality to accelerate the searching for nearest neighbors in a high dimensional space. The *k*M*k*NN algorithm has two stages. In the buildup stage, instead of using complex tree structures such as metric trees, *k*d-trees, or ball-tree, *k*M*k*NN uses a simple *k*-means clustering method to preprocess the training dataset. In the searching stage, given a query object, *k*M*k*NN finds nearest training objects starting from the nearest cluster to the query object and uses the triangle inequality to reduce the distance calculations. Experiments show that the performance of *k*M*k*NN is surprisingly good compared to the traditional *k*-NN algorithm and tree-based *k*-NN algorithms such as *k*d-trees and ball-trees. On a collection of 20 datasets with up to 10^6 records and 10^4 dimensions, *k*M*k*NN shows a 2-to 80-fold reduction of distance calculations and a 2- to 60-fold speedup over the traditional *k*-NN algorithm for 16 datasets. Furthermore, *k*M*k*NN performs significant better than a *k*d-tree based *k*-NN algorithm for all datasets and performs better than a ball-tree based *k*-NN algorithm for most datasets. The results show that *k*M*k*NN is effective for searching nearest neighbors in high dimensional spaces.

I. Introduction

The *k*-nearest neighbor (*k*-NN) algorithm is widely used in many areas such as pattern recognition, machine learning, and data mining. The traditional *k*-NN algorithm is called a lazy learner, as the buildup stage is cheap but the searching stage is expensive — the distances from a query object to all the training objects need to be calculated in order to find nearest neighbors for the query object [24]. One advantage of the traditional *k*-NN algorithm is that the running time is sublinear to *k*. For example, given a training set with *n* objects and *m* dimensions, if we maintain a max-heap structure for a set of *k* current nearest training objects, then the time complexity of the traditional *k*-NN algorithm is $O(mn\lg k)$ for a single query.

In 2D or 3D space, graph-based searching methods such as Voronoi diagram [17] and proximity graph [21] are efficient in searching for nearest neighbors, but it is very hard to extend these methods to higher dimensions. A number of spatial searching methods such as ball-trees [18], *k*d-trees [4], metric-trees [5] [22], quadtree [16], and R-trees [11] have been proposed to efficiently reduce the distance calculations and find exact nearest neighbors in higher dimensions. These methods iteratively divide training objects and build tree

(phone: 208-467-8769; fax: 208-467-8676; xwang@nnu.edu)..

structures using criteria such as absolute coordinates and relative distances, so that a query object needs to check distances with only a limited number of training objects instead of the whole dataset. One problem for these methods is that when the dimensionality of a dataset is high, most of the training objects in the data structures will end up being evaluated and the searching efficiency is no better to or even worse than the traditional *k*-NN algorithm [10] [18], especially for large *k* values.

Due to the difficulty of accelerating the *k*-NN algorithm in high dimensional space, some methods have focused on finding approximate answers. For example, the hashing method from [9] and the priority queue based method from [3] achieved a speedup of several fold over the traditional *k*-NN by outputting *k* neighbors within $(1+\varepsilon)$ of the true nearest neighbor distances. Hart [13] and Wilson [23] used techniques called condensing and editing to reduce objects from the dataset and accelerate the searching for nearest neighbors.

In this paper, we present a new algorithm called *k*M*k*NN (*k*-Means for *k*-Nearest Neighbors) to efficiently search exact *k* nearest training objects for a query object. The *k*M*k*NN algorithm incorporates two simple methods, the *k*-means clustering and the triangle inequality, into the nearest neighbors searching and achieves good performance compared to other algorithms. The basic idea is that we first classify training objects into different clusters regardless of the classes of training objects, and then for a given query object *q*, we use the triangle inequality to avoid distance calculations for some training objects in the clusters that are far from *q*.

The *k*M*k*NN algorithm has two stages. In the buildup stage, we separate the dataset into clusters and record the distance from each training object to its closest cluster center. In the searching stage, we first calculate the distances from a query object *q* to all cluster centers. Then we visit each training object starting from the nearest cluster to *q* and maintain a set of *k* current nearest objects and the largest distance $d_{\max}$ for the *k* nearest objects. For a training object *p* with its closest cluster center *c*, since we have the triangle inequality for the distances of *p*, *q*, and *c* such that $\|q - c\| < \|p - c\| + \|p - q\|$, if $d_{\max} < \text{abs}(\|q - c\| - \|p - c\|)$, then $d_{\max} < \|p - q\|$. Then we do not need to explicitly calculate the distance $\|p - q\|$.

Figure 1 shows a scenario of the *k*M*k*NN algorithm with a two-classes dataset. One class is shown as triangles, the other class is shown as pluses, and the query object is shown as a black circle. The dataset is separated into 6 clusters and most of the query objects' nearest neighbors are in the bottom-right cluster. In the searching stage, *k*M*k*NN first calculates the distances from the query object to all cluster centers. Then it searches *k* nearest training objects starting from the bottom-right cluster and uses the triangle inequality to avoid distance calculations for some training objects in some clusters such as the top-left one.

If the training objects can be well separated into clusters, then the *k*M*k*NN algorithm should be able to significantly reduce the distance calculations and accelerate searching for nearest training objects by using the triangle inequality. If the training objects are uniformly distributed or condensed together and there is no good way to build well separated clusters (for example, a uniformly distributed random dataset), then the triangle inequality may not apply and *k*M*k*NN may end up calculating the distances to all the training objects. It should be noted, though, that since the number of clusters is much smaller than the total number of training objects, the cost of extra distance calculations from the query object to the cluster centers is small compared to the total number of distances calculations. So the worst-case performance of *k*M*k*NN will be comparable to the traditional *k*-NN.

We tested the *k*M*k*NN algorithm on a collection of 20 datasets. The datasets, collected from the UCI Machine Learning Repository [8] and the National Cancer Institute, have up to 10^6 objects and 10^4 dimensions. We conducted three experiments on the datasets and all of them

used a small k value 9 and a large k value 101. In the first experiment, we compared kMkNN to the traditional k-NN algorithm on the reduction of distance calculations and the speedup of running time. The results show a 2- to 80-fold reduction in distance calculations and a 2- to 60-fold speedup in running time for 16 datasets. The performance of 2 worst datasets is comparable to the traditional k-NN algorithm (less than 5% of the slowdown of the running time). In the second experiment, we compare kMkNN to the kd-tree based algorithm on the speedup over the traditional k-NN for all datasets. The results show that kMkNN performs much better. In the last experiment, we compare to the ball-tree based algorithm [18] on the overall running time (the breakdown of the buildup and searching time is not available for the ball-tree based program we obtained [18]) and kMkNN shows better performance for most of datasets. All three experiments show that the kMkNN algorithm can effectively reduce the distance calculations and accelerate the searching for nearest neighbors in high dimensional spaces. It can be considered as an alternative for existing tree-based exact k-NN algorithms.

II. METHODS

Assume there are n objects p_i for ($1 \le i \le n$) in a training dataset and each object p_i has m dimensions. We present a new algorithm kMkNN (k-Means for k-Nearest Neighbors) that searches for exact k nearest neighbors to a query object q, shown as follows:

Algorithm 1

The kMkNN algorithm

The BUILDUP stage—Input: n training objects p_i for ($1 \le i \le n$); each with m dimensions.

Output: kc cluster centers c_j for ($1 \le j \le kc$) with the assignment of each object p_i to its nearest cluster.

1. Set $kc = s\sqrt{n}$, where $s > 0$.
2. Classify the n objects into kc clusters using a k-means clustering algorithm.
3. Calculate and record the distance d_{ij} from each object p_i to its nearest cluster center c_j.
4. For each cluster center c_j, sort the distances d_{ij} in descending order for all the training objects associated to c_j.

The SEARCHING stage—Input: kc cluster centers c_j for ($1 \le j \le kc$) with the assignments of each object p_i to its nearest center c_j; a query object q; the number of nearest neighbors k.

Output: A set of k nearest training objects to q.

1. Initialize the set of k nearest objects of q by using a maximum distance as the k distances. Set the maximum distance in the set of k current nearest objects to $d_{\max}$.
2. Calculate the distances $\|q - c_j\|$ for all cluster centers c_j that ($1 \le j \le kc$) and sort the distances in ascending order.
3. For each cluster c_j from the nearest to the farthest to q, do the step 4.
4. For each object p_i in the cluster c_j from the farthest to the nearest to the cluster center c_j:

if $d_{\max} \leq \|q - c_j\| - \|p_i - c_j\|$, go to step 3 to visit the next cluster;

otherwise, calculate the distance $\|q - p_i\|$ from q to p_i and if $d_{\max} > \|q - p_i\|$, remove the object with distance $d_{\max}$ from the set of k current nearest objects and insert p_i with the distance $\|q - p_i\|$; update $d_{\max}$.

The kMkNN algorithm runs in two stages. In the buildup stage, we separate the n training objects into kc clusters using a k-means clustering algorithm. After the clustering, we record the distance from each object to its nearest cluster center and sort all the distances for each cluster in a descending order. In the searching stage, we first initiate the distances of k nearest neighbors to a maximum value and calculate and sort the distances $\|q - c_j\|$ from q to each cluster center c_j for ($1 \leq j \leq kc$). Then starting from the nearest cluster to q, we use the triangle inequality to check each object p_i in a cluster c_j. If $d_{\max} \leq \|q - c_j\| - \|p_i - c_j\|$, where $d_{\max}$ is the maximum distance in the set of k current nearest objects to q, then we skip the whole cluster c_j. Otherwise, we calculate $\|q - p_i\|$ and if $\|q - p_i\| < d_{\max}$, we remove the object with the distance $d_{\max}$ from the set of k current nearest objects, insert p_i, and update $d_{\max}$. After we check all the clusters c_j for ($1 \leq j \leq kc$), the set of k current nearest objects contains the k nearest neighbors to q among all training objects p_i for ($1 \leq i \leq n$).

In the searching stage, given a query object q, we have two assumptions on how to visit the clusters and the objects in the clusters: (a) it is most likely that we can find most of the k nearest training objects to q in only a few nearest clusters to q; (b) for each object in a cluster other than the cluster closest to q, it is more likely that the distance from the object to the cluster center is shorter than the distance from q to the cluster center.

Assumption (a) is obvious, so in the algorithm, we visit each cluster from the nearest to the farthest to q after we calculate the distance $\|q - c_j\|$ from q to each cluster center c_j for ($1 \leq j \leq kc$) and sort the distances.

Assumption (b) does not hold for the cluster nearest to q (for example, the lower right cluster in Figure 1), but it does hold for other clusters (for example, the clusters other than the lower right one in Figure 1). The objects in those clusters will have shorter distances to their corresponding centers than the distances from q to those cluster centers, that is, q can be considered as an outlier for the majority of clusters.

Based on assumption (b), we can use the triangle inequality to reduce unnecessary distance calculations from some training objects p_i to q. For the cluster nearest to q, we mostly likely will check all objects and obtain a set of k current nearest objects with a small $d_{\max}$. For other clusters, say, c_j, most likely we have $\|q - c_j\| > \|p_i - c_j\|$, that is, q is farther from c_j than a training object p_i associated with c_j. Also based on the triangle inequality, we have $\|q - c_j\| - \|p_i - c_j\| < \|q - p_i\|$. If $d_{\max} \leq \|q - c_j\| - \|p_i - c_j\|$, then we have $d_{\max} \leq \|p_i - q\|$ and we do not need to calculate the distance $\|p_i - q\|$. Note that we do not use $d_{\max} \leq \text{abs}(\|q - c_j\| - \|p_i - c_j\|)$, since it is rare to have $\|p_i - c_j\| > \|q - c_j\|$, in which case q is closer to c_j than p_i for a non-nearest cluster to q, so we check $d_{\max} < \|q - c_j\| - \|p_i - c_j\|$ only in the implementation.

To further reduce distance calculations, for each cluster, we check from the farthest training object associated with the cluster center to the nearest one. The reason is that, first of all, for any cluster c_j other than the nearest one, generally $\|q - c_j\| > \|p_i - c_j\|$ for any object p_i associated with c_j. Given two objects p_{i1} and p_{i2} associated with c_j, if $\|p_{i1} - c_j\| > \|p_{i2} - c_j\|$ (i.e. p_{i1} is farther from c_j than p_{i2}), we have $\|q - c_j\| - \|p_{i1} - c_j\| < \|q - c_j\| - \|p_{i2} - c_j\|$ and p_{i1} is more likely to be a candidate of nearest neighbors to q than p_{i2}, so in the buildup stage we sort the distances from all training objects to their cluster centers in a descending order and in the searching stage we check the objects in that order.

By checking training objects from the farthest one to the nearest one for a cluster, we can even avoid checking the triangle inequality for some objects. For example, for any cluster c_j other than the one nearest to q, if we have $d_{\max} < \|q - c_j\| - \|p_i - c_j\|$ for a training object p_i, then we can skip the remaining training objects in c_j and move to the next cluster. Since we have $\|p_i - c_j\| > \|p_{ix} - c_j\|$ for any of the remaining objects p_{ix} in the cluster c_j, $\|q - c_j\| - \|p_i - c_j\| < \|q - c_j\| - \|p_{ix} - c_j\|$.

To efficiently obtain the largest distance $d_{\max}$ from the set of k current nearest objects to q, we can build a max-heap for the k distances that the root object has the largest distance (i.e. $d_{\max}$) to q. Every time when we find a new training object with a smaller distance to $d_{\max}$, we can efficiently remove the root object and insert new object in $O(\lg k)$ time. During the initialization, we set all the k values to a same maximum value, so the initial heap is built automatically.

One more issue is to choose the number of clusters kc. If the kc is not chosen carefully, then the k-means clustering may yield poor results. For kMkNN, the purpose of the clustering process is to separate the dataset into groups, so we may reduce the number of distance calculations in the searching stage. If we use a small number of kc, then each cluster will have many training objects and if the clusters are not separated very well, then the query object may be "close" to most of clusters, so kMkNN may result in a large number of distance calculations and the running time will increase. On the other hand, if we use a large number of kc, although each cluster will have a few training objects and we may have fewer distance calculations, the running time may also increase since a large number of kc have high extra costs.

Since it is very likely that we need to calculate distances from q to all objects in the cluster nearest to q and it is unlikely that we need to calculate distances from q to objects in other clusters, if we generate $kc = O(\sqrt{n})$ clusters with $O(\sqrt{n})$ objects in each cluster and if the number of nearest neighbors $k < \sqrt{n}$, then in the searching stage, kMkNN calculates distances to the training objects in only a few of the nearest clusters and finds k nearest neighbors. Calculating the distances from a query object q to all cluster centers takes $O(m\sqrt{n})$, sorting of all the clusters from the nearest to the farthest to q takes $O(\sqrt{n}\lg n)$, checking triangle inequality for all training objects (in the worst case) takes $O(n)$, and calculating distances from q to all training objects in a few nearest clusters takes $O(m\sqrt{n})$ (in the worst case; note kMkNN may calculate the distances from q to most but not all training objects in those nearest clusters – – the actual number of calculations is determined from the triangle inequality), so if $m << n$, then we may achieve a near optimal performance by choosing $kc = \sqrt{n}$. In practice the dataset is never ideal and many factors will influence performance.

The ideal complexity analysis above does not consider the constant factor associated with each time complexity term, so in the implementation we set $kc = O(\sqrt{n}) = s\sqrt{n}$ by introducing a constant $s > 0$. Our experiments in Section 3 show that kMkNN achieves good running time for most of datasets when $s = 2.0$, but the optimal value of s may vary depending on a specific dataset and the number k.

We note that the searching stage of the kMkNN algorithm adds a small overhead when comparing to the traditional k-NN algorithm in the worst case. Given $O(\sqrt{n})$ clusters, calculating and sorting the distances from q to all cluster centers takes $O(m\sqrt{n}\lg mn)$ time, and comparing $d_{\max}$ with $\|q - c_j\| - \|p_i - c_j\|$ for all training objects using the triangle

inequality takes $O(n)$ time in the worst case, so the total extra cost is $O\left(m\sqrt{n}\lg mn+n\right)$, which is less than the $O(mn)$ running time for the traditional k-NN. The small overhead mean that even in the worst case scenario when the triangle inequality does not work and we end up calculating all the distances, the running time of kMkNN is still close to the traditional k-NN.

In the buildup stage, the goal is to split the dataset into clusters, so any clustering algorithm will work. We use Lloyd's algorithm [19] in the implementation, which is a simple heuristic algorithm that iteratively converges to a local minimum. There are faster k-means clustering algorithms available [6] [7] and we can use these algorithms to speed up the buildup stage of kMkNN.

There is no guarantee that Lloyd's algorithm [19] will converge to the global minimum, that is, minimizing the sum of all the distances from all the objects to their nearest centers — an NP-hard problem in general [1] and an $O(n^{mk+1}\lg n)$ problem, given m dimensions and k clusters [14]. Ideally, if the clustering algorithm converges to a local minimum closer to the global minimum, then each cluster is relatively denser and we may reduce the number of distance calculations in the searching stage. Some k-means clustering algorithms [2] [15] have shown to be able to produce better results than Lloyd's algorithm, but for this paper, we already achieve excellent performance by using the naive Lloyd's algorithm. It will be interesting to see how performance can be further improved by integrating these algorithms.

III. EXPERIMENTS AND DISCUSSION

A. Datasets

We test the kMkNN algorithm on a collection of 20 datasets, as listed in Table 1. The C++ code of kMkNN and all the processed datasets are available upon request.

18 datasets (*abalone*, *arcene* [12], *car*, *chess*, *dorothea* [12], *gisette* [12], *image*, *ipums* [20], *isolet*, *kddcup99*, *letter*, *multiple*, *musk1*, *musk2*, *poker*, *satalog*, *semeion*, *spambase*) are from the UCI Machine Learning Repository. For the *chess*, we use the dataset *King-Rook vs. King*. For the *dorothea*, we use the first 654 features as each object has different number of features. For the *ipums*, we use the *ipums.la.97* dataset and predict *farm status*. For the *kddcup99*, we use the 10% subset. For the *musk1* and *musk2*, we use the *clean1* and *clean2* in the *Musk (Version 2)*.

2 other datasets (*ds1.10pca* and *ds1.100pca*) are derived from dataset *ds1* in the Open Compound Database provided by the National Cancer Institute (NCI), whereas they are the linear projection of the top 10 and 100 dimensions by principle component analysis (PCA).

B. Experiments

We compared the performance of kMkNN to the traditional k-NN, kd-tree based, and ball-tree based algorithms in three experiments. All experiments used a 10-fold cross validation. We tested the performance of kMkNN on a small k value 9 and a large k value 101. Furthermore, we used s = 2.0, which generates $kc=2.0\sqrt{n}$ clusters. We tested s = 1.0, 2.0, and 3.0 and s = 2.0 shows the best overall performance.

In the first experiment, we compared kMkNN to the traditional k-NN algorithm on the reduction of distance calculations and the speedup of running time. The results are shown in Tables II and III. For the distance calculations and the running time, we recorded the total number of distance calculations and the total running time in all cross validations. For the running time of the kMkNN algorithm, we ignored the buildup time and recorded the

searching time. For the traditional *k*-NN algorithm, the total number of distance calculations is $0.9n^2$ for a 10-fold cross validation test with a dataset with *n* objects, so the number of distance calculations is the same for any *k* value.

In the second experiment, we compared *k*M*k*NN to the *k*d-tree based algorithm on the speedup over the traditional *k*-NN. The results are shown in Table IV. We used the *k*d-tree based algorithm implemented in MATLAB and the speedup was calculated by comparing the running time to the traditional *k*-NN algorithm implemented in MATLAB (labeled "exhaustive"). For both the *k*d-tree based and the traditional *k*-NN algorithms in MATLAB, we ignored the classifier buildup time and compared the searching/classifying time only, as the classifier buildup time in MATLAB may have some extra costs.

In the last experiment, we compared to the ball-tree based algorithm [18] on the overall running time (buildup + searching time). The results are shown in Table V. For the ball-tree based algorithm, since only an executable version is available from [18], we could not break down the buildup and searching time. So we compared the total running time.

C. Results and Discussion

In the first experiment, the comparison to the traditional *k*-NN algorithm in Tables II and III shows that the *k*M*k*NN algorithm can effectively reduce the distance calculations as well as accelerate the *k*-nearest neighbor searching, whereas the worst case performance is still close to the traditional *k*-NN algorithm. Both tables show that *k*M*k*NN effectively reduces the distance calculations by 2- to 80-fold and reduces the running time by 2- to 60-fold in 16 datasets, whereas *k*M*k*NN show a slight downgrade of performance (less than 5%) in two datasets (*gisette* and *semeion*). Comparing $k = 9$ and 101, both the reduction of distance calculations and the speedup of the running time are decreased slowly ($k = 101$ is less than 2-fold worse than $k = 9$), which shows that the performance of *k*M*k*NN is robust with *k* increases.

When the number of objects increases and the number of clusters *kc* increases, generally the clusters tend to be more condensed and we can better use the triangle inequality, so we see a slight increase of the speedup in both criteria. For example, the datasets *kddcup99* and *poker*, which have 5×10^5 and 10^6 objects each, show 70-fold and 30-fold speedup in the running time. When the dimension size increases, generally the clusters tend to be more expanded, so we see a slight decrease of the speedup in both criteria. But *k*M*k*NN still performs well for datasets with high dimensions. For example, the dataset *arcene*, which has 10^4 dimensions, shows over 2-fold speedup in the running time.

For the datasets *gisette* and *semeion*, the performance of *k*M*k*NN is not as good as the traditional *k*-NN algorithm. The main reason is that the objects in both datasets form single condensed clusters instead of spreading out. For example, the dataset *gisette* is for classifying two highly confusable digits '4' and '9' and the objects in the dataset are similar to each other. Although it has 5000 dimensions, it forms a single condensed cluster due to the similarity of objects. Although in the buildup stage *k*M*k*NN separates the objects into many clusters, for most of query objects the distances from the query object to the cluster centers and the distances from the training objects to the cluster centers are close, so the triangle inequality does not work well and we end up calculating most of the distances. We also see the similar worst performance in the simulated datasets with uniformly distributed random objects. Still, as shown in Tables II and III, in the worst case scenario, the performance of *k*M*k*NN is close to the traditional *k*-NN algorithm due to the small overhead.

In the second experiment, Table IV shows that *k*M*k*NN is on average 3-40 times faster than the *k*d-tree based algorithm for all 20 datasets. The *k*d-tree based algorithm is usually used

for searching low dimensional datasets with small *k* values. Even when we compare to the traditional *k*-NN algorithm, for the 20 datasets we used, the *k*d-tree based algorithm shows better performance in only a few datasets but shows 4-5 times worse performance in about half of the datasets. It may suggest that the *k*d-tree based algorithm is not suitable for searching nearest neighbors in high dimensional space and *k*M*k*NN or other spatial searching methods should be used instead.

In the third experiment, Table V shows that *k*M*k*NN performs better than the ball-tree based algorithm for most of datasets. The ball-tree algorithm [18] is one of the spatial searching methods that improves the *k*d-tree based algorithm and is efficient for searching nearest neighbors in high dimensional space. Although we could not obtain the source code for the ball-tree algorithm and can compare only the total running time, we can still see from the Table V that *k*M*k*NN has a better overall performance when combing the buildup and searching time. It shows that *k*M*k*NN can be considered as an alternative for these spatial searching methods in the nearest neighbors searching.

There are two advantages of the *k*M*k*NN algorithm over the tree-based spatial searching algorithms. The first advantage is that *k*M*k*NN has a small overhead in searching nearest neighbors. Given a dataset of *n* training objects, generally a tree-based algorithm has $O(\lg n)$ levels and $O(n)$ nodes, where each node has some extra costs for boundary checking. If a search for nearest neighbors involves only a few nodes at each level, then a total of $O(\lg n)$ nodes will be visited and the algorithm can be much better than *k*M*k*NN. But if a search involves most of nodes in each level, then a total of $O(n)$ nodes will be visited and the algorithm can be much worse than *k*M*k*NN. Even when the *k* nearest objects are eventually from a few nodes, too much extra time may be wasted on walking through the internal/leaf nodes, so the algorithm can perform badly due to the extra costs for each node. *k*M*k*NN is like a tree with two levels, one root node and *kc* leaf nodes. There are more operations inside each leaf node, but the whole data structure is flat so there is no extra cost for walking through the internal nodes.

Furthermore, the implementation of the *k*M*k*NN algorithm uses arrays instead of pointers, which are used in the tree-based algorithms, so the constant associated with the time complexity of *k*M*k*NN is small. When both *k*M*k*NN and a spatial searching algorithm have the same number of steps for a searching, the *k*M*k*NN will run faster in practice because of the small constant associated with each step.

The second advantage is *k*M*k*NN has a better way of organizing objects. The tree-based algorithms use various criteria such as individual dimensions to separate the objects. There are two potential problems. One is that these criteria may not match the way the objects are distributed or queried, so the objects may not separate or be queried well. For example, one reason why the *k*d-tree method is worse than some other tree-based methods in high dimensions is that the *k*d-tree method uses the dimensions to separate objects, but the actual objects are usually organized by their relative distances but not by their dimensions. The other problem is that these algorithms always do binary separation each time, but maybe the objects are better separated in three or more groups directly. *k*M*k*NN overcomes the first problem by using the criterion of object distances, to separate the objects and overcomes the second problem by separating the objects into multiple clusters directly. It may be possible to build a new tree-based algorithm that uses the distance criteria and allows multiple children per parent node. If a dataset contains many small clusters and we can organize the clusters in a hierarchical manner, then a multiple level *k*-means clustering algorithm may be able to further improve the performance of *k*M*k*NN.

IV. CONCLUSION

We present a new algorithm *k*M*k*NN for the nearest neighbors searching problem. The algorithm implements two simple methods: the *k*-means clustering and the triangle inequality, into the nearest neighbors searching and the experiments show that the algorithm performs well and can be considered as alternative choice for searching nearest neighbors in high dimensional spaces.

One interesting future work is to integrate better *k*-means clustering algorithms to *k*M*k*NN that generate better clusters than Lloyd's algorithm and see if the performance of *k*M*k*NN can be improved further. Another work is to see if we can determine an optimal number of clusters based on the number of training objects, the number of dimensions, and the distribution of the training objects. One further future work is that, since *k*M*k*NN and the spatial searching methods may work well on different datasets, it will be interesting to analyze these datasets and try to find correlation between the distribution of the objects in the datasets and the performance of various algorithms.

Acknowledgments

We thank Dr. Andrew Moore at Carnegie Mellon University for providing the ball-tree program. We thank Erik Lawrence for collecting some datasets and running some experiments.

This work was supported by NIH Grant #P20 RR0116454.

References

[1]. Aloise D, Deshpande A, Hansen P, Popat P. NP-hardness of Euclidean sum-of-squares clustering. Machine Learning. 2009; vol. 75(no. 2):245–248.

[2]. Arthur, D.; Vassilvitskii, S. *k*-means++: the advantages of careful seeding. Proc. 18th annual ACM-SIAM symposium on Discrete algorithms; 2007. p. 1027-1035.

[3]. Arya S, Mount DM, Netanyahu NS, Silverman R, Wu AY. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. Journal of the ACM. 1998; vol. 45(no. 6):891–923.

[4]. de Berg, M.; Cheong, O.; van Kreveld, M.; Overmars, M. Computational geometry: algorithms and applications. 3rd ed. Springer-Verlag; New York: 2008.

[5]. Ciaccia, P.; Patella, M.; Zezula, P. M-tree: an efficient access method for similarity search in metric spaces; Proc. 23rd International Conference on VLDB; 1997. p. 426-435.

[6]. Elkan, C. Using the triangle inequality to accelerate *k*-means. Proc. 12th International Conference on Machine Learning; 2003. p. 147-153.

[7]. Frahling, G.; Sohler, C. A fast k-means implementation using corsets. Proc. 22nd annual symposium on Computational geometry; 2006. p. 135-143.

[8]. Frank, A.; Asuncion, A. UCI Machine Learning Repository. University of California, School of Information and Computer Science; Irvine, CA: 2010. [http://archive.ics.uci.edu/ml]

[9]. Gionis, A.; Indyk, P.; Motwani, R. Similarity search in high dimensions via hashing. Proc. 25th International Conference on VLDB; 1999. p. 518-529.

[10]. Goodman, JE.; O'Rourke, J. Handbook of Discrete and Computational Geometry. 2nd ed. CRC Press; 2004. ch. 39.

[11]. Guttman, A. R-trees: a dynamic index structure for spatial searching. Proc. 3rd ACM SIGMOD International Conference on Management of Data; 1984. p. 47-57.

[12]. Guyon IM, Gunn SR, Ben-Hur A, Dror G. Result analysis of the NIPS 2003 feature selection challenge. Advances in Neural Information Processing Systems. 2004

[13]. Hart P. The condensed nearest neighbor rule. IEEE Trans. on Inform. Th. 1968; vol. 14:515–516.

[14]. Inaba, M.; Katoh, N.; Imai, H. Applications of weighted Voronoi diagrams and randomization to variance-based k-clustering. Proc. 10th ACM Symposium on Computational Geometry; 1994. p. 332-339.

[15]. Kanungo T, Mount DM, Netanyahu NS, Piatko CD, Silverman R, Wu AY. A local search approximation algorithm for k-means clustering. Computational Geometry: Theory and Applications. 2004; vol. 28:89–112.

[16]. Kim YJ, Patel JM. Performance comparison of the R-tree and the quadtree for kNN and distance join queries. IEEE Trans. on Knowledge and Data Engineering. 2010; vol. 22(no. 7):1014–1027.

[17]. Kolahdouzan, M.; Shahabi, C. Voronoi-based *k* nearest neighbor search for spatial network databases. Proc. 30th International Conference on VLDB; 2004. p. 840-851.

[18]. Liu T, Moore AW, Gray A. Efficient exact *k*-NN and nonparametric classification in high dimensions. Proc. of Neural Information Processing Systems. 2003

[19]. Lloyd SP. Least squares quantization in PCM. IEEE Trans. on Inform. Th. 1982; vol. 28(no. 2): 129–137.

[20]. Ruggles, S.; Sobek, M. Integrated public use microdata series. version 2.0. Historical Census Projects, University of Minnesota; Minneapolis: 1997.

[21]. Toussaint, GT. Proximity graphs for nearest neighbor decision rules: recent progress. Proc. 34th symposium on computing and statistics; 2002. p. 17-20.

[22]. Uhlmann JK. Satisfying general proximity/similarity queries with metric trees. Information Processing Letters. 1991; vol. 40:175–179.

[23]. Wilson DL. Asymptotic properties of nearest neighbor rules using edited data. IEEE Trans. Syst. Man. Cyberne. 1972; vol. 2:408–420.

[24]. Wu X, Kumar V, Quinlan JR, Ghosh J, Yang Q, Motoda H, McLachlan GJ, Ng A, Liu B, Yu PS, Zhou Z, Steinbach M, Hand DJ, Steinberg D. Top 10 algorithms in data mining. Knowl. Inf. Syst. 2008; vol. 14:1–37.

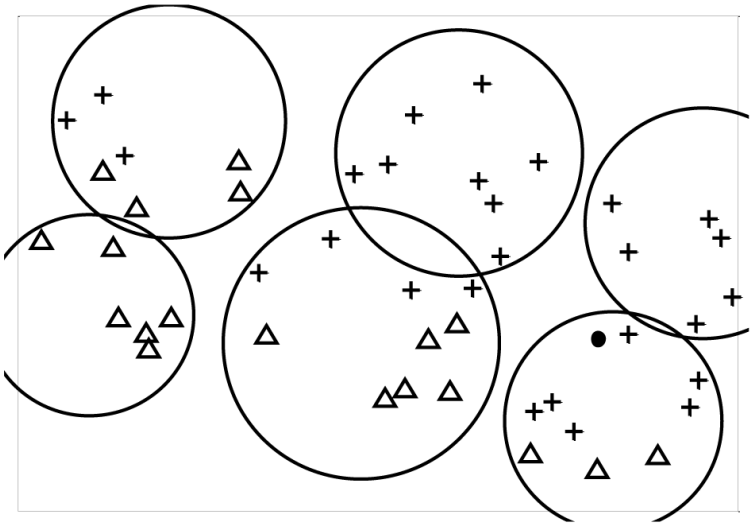

Fig. 1.

A scenario of the *k*M*k*NN algorithm. The plus and triangle signs denote two classes of objects and the black dot denotes the query object. The *k*M*k*NN algorithm first classifies objects into clusters and then finds the *k*-nearest neighbors for a query object starting from the nearest cluster.

TABLE I

The 20 Datasets

Dataset	#Objects	#Dimensions
abalone	4177	8
arcene	900	10000
car	1728	6
chess	28056	6
dorothea	1950	770
ds1.10pca	26733	10
ds1.100pca	26733	100
gisette	13500	5000
image	2310	19
ipums	70187	60
isolet	7797	617
kddcup99	494021	41
letter	20000	16
multiple feat.	2000	659
musk clean1	476	166
musk clean2	6598	166
poker	1000123	10
satalog	6435	36
semeion	1593	256
spambase	4601	57

TABLE II

The Number of Distance Calculations of the *KMK*NN and the Traditional *K*-NN Algorithms.

Dataset	Distance Calculations *kMk*NN reduction		
	k*-NN	***k* = 9**	***k* = 101**
abalone	1.6×10^7	16.3	11.0
arcene	7.3×10^5	2.6	2.3
car	2.7×10^6	5.4	2.0
chess	7.1×10^8	25.8	13.7
dorothea	3.4×10^6	6.7	4.5
ds1.10pca	6.4×10^8	19.1	7.7
ds1.100pca	6.4×10^8	2.7	1.4
gisette	2.3×10^7	1.0	1.0
image	4.8×10^6	13.2	6.2
ipums	4.4×10^9	85.1	63.9
isolet	5.5×10^7	1.4	1.2
kddcup99	2.2×10^{11}	80.4	72.2
letter	3.6×10^8	14.8	6.0
multiple feat.	3.6×10^6	7.0	4.2
musk clean1	2.0×10^5	1.8	1.3
musk clean2	3.9×10^7	5.3	2.9
poker	9.0×10^{11}	53.5	28.3
satalog	3.7×10^7	8.0	5.5
semeion	2.3×10^6	1.0	1.0
spambase	1.9×10^7	15.2	9.6

* For the traditional *k*-NN algorithm, the number of distance calculations is the same for different *k*.

TABLE III

The Running Time of the *KMKNN* and the Traditional *K*-NN Algorithms

Dataset	Traditional *k*-NN (s)		Running Time *kMkNN* Speedup	
	***k* = 9**	***k* = 101**	***k* = 9**	***k* = 101**
abalone	1.8	1.8	22.6	12.2
arcene	10.7	10.7	2.7	2.3
car	0.2	0.3	4.4	2.6
chess	77.3	76.0	30.9	22.5
dorothea	3.4	3.4	7.3	4.9
ds1.10pca	91.3	91.3	33.3	20.0
ds1.100pca	182.4	175.4	4.5	2.2
gisette	158.2	157.9	1.0	1.0
image	0.6	0.6	17.7	6.8
ipums	977.9	981.6	61.3	53.5
isolet	52.8	52.9	1.5	1.3
kddcup99	38200.0	38239.1	59.1	55.0
letter	50.2	50.2	24.8	14.0
multiple feat.	3.5	3.5	7.9	4.6
musk clean1	0.1	0.1	1.7	1.3
musk clean2	13.2	13.2	7.3	3.9
poker	130220.3	130357.9	60.1	49.9
satalog	5.9	6.0	13.7	9.6
semeion	0.9	0.9	1.2	1.1
spambase	3.5	3.5	18.6	12.2

TABLE IV

The Speedup of the Running Time Over The Traditional *K*-NN Algorithm for the *K*M*K*NN and the *K*d-Tree Algorithms

Dataset	*k* = 9		*k* = 101	
	***k*M*k*NN**	***k*d-Tree**	***k*M*k*NN**	***k*d-Tree**
abalone	18.5	3.53	13.5	2.15
arcene	2.7	0.23	2.6	0.24
car	4.6	1.51	3.4	1.20
chess	27.9	7.74	20.6	4.01
dorothea	7.0	0.30	4.7	0.26
ds1.10pca	28.6	2.93	14.8	1.17
ds1.100pca	3.6	0.30	1.9	0.23
gisette	1.0	0.23	1.0	0.23
image	14.6	0.90	7.3	0.60
ipums	42.8	11.63	39.0	6.42
isolet	1.4	0.22	1.2	0.22
kddcup99 10%	44.8	1.07	46.3	1.05
letter	22.5	0.95	10.8	0.48
multiple feat.	7.4	0.22	4.4	0.22
musk v2 clean1	2.3	0.23	1.3	0.26
musk v2 clean2	6.1	0.22	3.3	0.21
poker	37.6	18.48	33.7	7.36
satalog	12.0	0.48	7.9	0.40
semeion	1.0	0.21	1.0	0.22
spambase	17.9	0.78	11.5	0.70

TABLE V

The Overall Running Time of the *K*M*K*NN and the Ball-Tree Algorithms

Dataset	*k* = 9 (s)		*k* = 101 (s)	
	***k*M*k*NN**	**Ball-Tree**	***k*M*k*NN**	**Ball-Tree**
abalone	6.3	3.2	4.6	4.8
arcene	183.2	1865.3	202.4	2320.6
car	0.9	1.3	0.8	2.5
chess	166.1	40.7	180.0	78.5
dorothea	41.3	136.4	49.1	197.7
ds1.10pca	131.9	121.6	150.8	236.0
ds1.100pca	927.0	5913.6	912.9	8225.9
gisette	1965.5	47435.9	1992.5	48432.2
image	2.3	4.0	2.3	7.0
ipums	2790.1	823.1	2887.6	1375.7
isolet	694.6	8548.3	646.2	8665.7
kddcup99 10%	2435.1	>86400	6162.4	>86400
letter	125.4	129.1	120.5	249.6
multiple feat.	54.5	139.4	53.0	215.9
musk v2 clean1	0.8	6.7	0.8	8.7
musk v2 clean2	92.7	286.7	82.9	465.2
poker	124089.9	44437.1	127338.6	83055.0
satalog	31.0	41.9	30.0	62.6
semeion	11.4	121.1	11.3	129.3
spambase	8.5	22.3	7.3	31.6