Winner-Update Algorithm for Nearest Neighbor Search

Yong-Sheng Chen†‡ Yi-Ping Hung†‡ Chiou-Shann Fuh‡

†Institute of Information Science, Academia Sinica, Taipei, Taiwan
‡Dept. of Computer Science and Information Engineering, National Taiwan University, Taiwan
Email: hung@iis.sinica.edu.tw

Abstract

This paper presents an algorithm, called the winner-update algorithm, for accelerating the nearest neighbor search. By constructing a hierarchical structure for each feature point in the l_p metric space, this algorithm can save a large amount of computation at the expense of moderate preprocessing and twice the memory storage. Given a query point, the cost for computing the distances from this point to all the sample points can be reduced by using a lower bound list of the distance established from Minkowski's inequality. Our experiments have shown that the proposed algorithm can save a large amount of computation, especially when the distance between the query point and its nearest neighbor is relatively small. With slight modification, the winner-update algorithm can also speed up the search for k nearest neighbors, neighbors within a specified distance threshold, and neighbors close to the nearest neighbor.

1. Introduction

Nearest neighbor rule has been widely applied in many fields, including template matching, pattern recognition, data compression, and information retrieval. In general, a fixed set of s sample points, $P = \{\mathbf{p}_i | i = 1, \ldots, s\}$ are given and preprocessing is performed if necessary. For each query point $\mathbf{q}$, the goal of nearest neighbor search is to find in P the point closest to $\mathbf{q}$. Let d denote the dimension of the underlying space. The computational complexity of the exhaustive search for finding the nearest neighbor is $O(sd)$. When s, d, or both are large, this process is very time-consuming.

In the literature, many methods have been proposed to speed up the computations of nearest neighbor search. Bentley [1] proposed a K-dimensional binary search tree method. They partitioned the space of sample points into hyper-rectangular buckets. Their search process for the closest point consists of a binary search of order $O(\log s)$ for a target bucket and a local search for the desired sample point in the target bucket and its neighboring buckets. This method is very efficient when d is small. However, its performance degrades exponentially with increasing d [4]. The reason is that more neighboring buckets are to be checked when d is higher. Thus, the number of sample points to be examined increases dramatically. Fukunaga and Narendra [2] also proposed a fast algorithm for finding k nearest neighbors. They divided the set of sample points into subsets and then divided each subset into subsets further. A tree structure can be constructed by repeating this process. Given a query point, they can then use a branch-and-bound search to efficiently find the closest point by using the tree structure.

Recently, Lee and Chen proposed a set of inequalities of the distance and applied to fast vector quantization [3]. By using these inequalities, they defined some test measures which require less computation than the distance do. For each sample point, its distance from the query point is larger than its test measures. If one of the test measures is larger than the minimum distance computed so far, this sample point can be excluded from further consideration. Hence, many distance computations can be saved. However, a lot of useless calculation will be performed until a sample point with small distance is examined. In the worst case, the distances are monotonically decreasing with respect to the examining order of the sample points. For such case, the computational cost will even be higher than that of using the exhaustive search.

In this paper, we propose an algorithm, called the winner-update algorithm, to efficiently find the nearest neighbor. Here, we choose the l_p norms as the distance function in the d-dimensional space. Given a query point, the straightforward way to find the nearest neighbor is to first compute the distances from the query point to all the points in the sample set and then choose the sample point which gives the smallest distance. To reduce the time required for computing the distances, our algorithm adopts the winner-update strategy which utilizes a lower bound list for each distance. In this paper, we use the lower bound list derived from Minkowski's inequality, which requires a hierarchical structure for each feature point. For many applications, the hierarchical structure for the sample points can be constructed beforehand, and only twice the memory storage is needed. The winner-update search strategy, which is a special case of branch-and-bound search strategy, can reduce a

0-7695-0750-6/00 $10.00 © 2000 IEEE

large amount of distance calculations by using these lower bounds. The examining order of the sample points will not affect the performance of the algorithm. With slight modification, the winner-update algorithm can also speed up the search for k nearest neighbors, neighbors within a specified distance threshold, and neighbors close to the nearest neighbor. Our experiments have shown that the proposed algorithm can save a large amount of computation, especially when the distance between the query point and its nearest neighbor is relatively small.

2. Fast Nearest Neighbor Search

2.1. Lower bounds of distance measure

For a point $\mathbf{z}$ in R^d space, $\mathbf{z} = (z_1, z_2, \ldots, z_d)$, its l_p norm is defined as $\|\mathbf{z}\|_p \equiv (\sum_{i=1}^{d} |z_i|^p)^{\frac{1}{p}}$. By using Minkowski's inequality, $\|\mathbf{x}+\mathbf{y}\|_p \leq \|\mathbf{x}\|_p + \|\mathbf{y}\|_p$, one can obtain $\|\mathbf{x}-\mathbf{y}\|_p \geq |\|\mathbf{x}\|_p - \|\mathbf{y}\|_p|$. For example, $d = 2$,

$$(|x_1 - y_1|^p + |x_2 - y_2|^p)^{\frac{1}{p}} \geq |(|x_1|^p + |x_2|^p)^{\frac{1}{p}} - (|y_1|^p + |y_2|^p)^{\frac{1}{p}}|,$$

$$|x_1 - y_1|^p + |x_2 - y_2|^p \geq |(|x_1|^p + |x_2|^p)^{\frac{1}{p}} - (|y_1|^p + |y_2|^p)^{\frac{1}{p}}|^p. \quad (1)$$

Without loss of generality, we assume that the dimension of the feature space, d, is equal to 2^L. For each point $\mathbf{x}$, we construct a $(L+1)$-level structure (from level L to level 0) by using the following equation:

$$x_i^{l-1} = (|x_{2i-1}^l|^p + |x_{2i}^l|^p)^{\frac{1}{p}}, \quad (2)$$

where the level number $l = L, \ldots, 1$ and $i = 1, \ldots, 2^{l-1}$. The 2^L elements, x_i^L, $i = 1, \ldots, 2^L$, at level L store the original point $\mathbf{x}$, while $\mathbf{x}^l = ({x^l}_1, {x^l}_2, \ldots, {x^l}_{2^l})$, $l = 0, \ldots, L-1$, are points in spaces of smaller dimensions, i.e., $2^0, 2^1, \ldots, 2^{L-1}$. In the following, $\mathbf{x}^l$ is referred to as the l-level projection of $\mathbf{x}$. Notice that twice the memory storage is needed to store this hierarchical structure, $\mathbf{x}^0, \mathbf{x}^1, \ldots, \mathbf{x}^L$, for each point $\mathbf{x}$.

For points $\mathbf{x}$ and $\mathbf{y}$, the following inequality between the distances calculated at the l-th and $(l-1)$-th levels can be obtained from Equations (1) and (2):

$$\begin{aligned}(\sum_{i=1}^{2^l} |x_i^l - y_i^l|^p)^{\frac{1}{p}} &= (\sum_{i=1}^{2^{l-1}} (|x_{2i-1}^l - y_{2i-1}^l|^p + |x_{2i}^l - y_{2i}^l|^p))^{\frac{1}{p}} \\ &\geq (\sum_{i=1}^{2^{l-1}} |(|x_{2i-1}^l|^p + |x_{2i}^l|^p)^{\frac{1}{p}} - \\ &\qquad (|y_{2i-1}^l|^p + |y_{2i}^l|^p)^{\frac{1}{p}}|^p)^{\frac{1}{p}} \\ &= (\sum_{i=1}^{2^{l-1}} |x_i^{l-1} - y_i^{l-1}|^p)^{\frac{1}{p}}.\end{aligned}$$

That is, $\|\mathbf{x}^{l-1} - \mathbf{y}^{l-1}\|_p \leq \|\mathbf{x}^l - \mathbf{y}^l\|_p$, $l = L, \ldots, 1$. As a result, we obtain an ascending list of the lower bounds (LBs) of the distance between points $\mathbf{x}$ and $\mathbf{y}$:

$$LB^0(\mathbf{x},\mathbf{y}) \leq LB^1(\mathbf{x},\mathbf{y}) \leq \ldots \leq LB^L(\mathbf{x},\mathbf{y})$$

where $LB^l(\mathbf{x},\mathbf{y}) \equiv \|\mathbf{x}^l - \mathbf{y}^l\|_p$ is the distance between $\mathbf{x}^l$ and $\mathbf{y}^l$, i.e., the l-level projection of $\mathbf{x}$ and $\mathbf{y}$ in 2^l-dimensional space. Notice that $LB^L(\mathbf{x},\mathbf{y})$ is the desired distance $\|\mathbf{x}-\mathbf{y}\|_p$. When the level number l is smaller, the computational cost of $LB^l(\mathbf{x},\mathbf{y})$ is smaller because the dimension at the l-th level is smaller.

2.2. Winner-update algorithm

Given a fixed set of sample points $P = \{\mathbf{p}_i | i = 1, \ldots, s\}$ and a query point $\mathbf{q}$ in R^{2^L} space, $(L+1)$-level structure for each point is first constructed by using Equation (2). Zero padding can be used if the dimension of the underlying space is not 2^L. The goal of nearest neighbor search is to find the point $\hat{\mathbf{p}}$ in P such that the distance measure $\|\hat{\mathbf{p}} - \mathbf{q}\|_p$ is minimum. For each sample point $\mathbf{p}_i$, if any of the lower bound $LB^l(\mathbf{p}_i, \mathbf{q})$, $l = 0, \ldots, L-1$, of the distance $\|\mathbf{p}_i - \mathbf{q}\|_p$ is larger than $\|\hat{\mathbf{p}} - \mathbf{q}\|_p$, $\mathbf{p}_i$ has no chance to be the nearest neighbor because $\|\mathbf{p}_i - \mathbf{q}\|_p \geq LB^l(\mathbf{p}_i, \mathbf{q}) > \|\hat{\mathbf{p}} - \mathbf{q}\|_p$. Consequently, the more-expensive calculation of the distance measure $\|\mathbf{p}_i - \mathbf{q}\|_p$ can be replaced by the less-expensive calculation of its lower bounds. The computational cost of the lower bounds is less than that of $\|\mathbf{p}_i - \mathbf{q}\|_p$ because $LB^l(\mathbf{p}_i, \mathbf{q})$, $l = 0, \ldots, L-1$, is the distance measure in the space with lower dimension (2^l), This is the reason why the computational cost can be saved by using the lower bound list.

In the following, we will describe the proposed winner-update algorithm for minimizing the number of lower bounds actually calculated. At first, we sort all the sample points $\mathbf{p}_i$, $i = 1, \ldots, s$, according to their 0-level projection, i.e., ${\mathbf{p}_i}^0$, and obtain a sorted list of 0-level projections. (Notice that the dimension at level 0 is 1.) Then, the temporary winner, $\hat{\mathbf{p}}$ is initially chosen to be the sample point whose 0-level projection, $\hat{\mathbf{p}}^0$, is closest to $\mathbf{q}^0$, the 0-level projection of the query point $\mathbf{q}$. This temporary winner can be found by using binary search (with complexity $O(\log s)$). Let P^0 be the set of all 0-level projections of sample points excluding those of sample points having been the temporary winner. At any time instant, if the 0-level projection of a sample point is in P^0, then it has not yet been considered as a competitor of the temporary winner. Next, the temporary runner-up $\tilde{\mathbf{p}}$ is defined to be the sample point whose 0-level projection, $\tilde{\mathbf{p}}^0$, is closest to $\mathbf{q}^0$ within P^0. This can be found by simply checking the neighbor(s) of $\hat{\mathbf{p}}^0$ in the sorted list of ${\mathbf{p}_i}^0$, $i = 1, \ldots, s$. The lower bound $LB(\hat{\mathbf{p}}, \mathbf{q})$ of the distance $\|\hat{\mathbf{p}} - \mathbf{q}\|_p$ is initialized to be $LB^0(\hat{\mathbf{p}}, \mathbf{q}) = \|\hat{\mathbf{p}}^0 - \mathbf{q}^0\|_p$. A variable for each $\mathbf{p}$, $l(\mathbf{p})$, is used to record the level where the lower bound of $\mathbf{p}$'s distance to $\mathbf{q}$ is calculated, which is initialized as 0. The temporary winner $\hat{\mathbf{p}}$ is then inserted into an empty heap data structure, in which the point with minimum lower bound is kept on the top. At each iteration, the temporary winner $\hat{\mathbf{p}}$ on top of the heap is selected to update its lower bound by calculating $LB^{l(\hat{\mathbf{p}})+1}(\hat{\mathbf{p}}, \mathbf{q})$ at the next level. The heap data structure is re-organized ("DownHeap") by moving down the point $\hat{\mathbf{p}}$ without violating the heap property. Then a new temporary winner $\hat{\mathbf{p}}$ on top of the heap is selected. If the distance $\|\tilde{\mathbf{p}}^0 - \mathbf{q}^0\|_p$ of the runner-up $\tilde{\mathbf{p}}$ at level 0 is smaller than $LB^{l(\hat{\mathbf{p}})}(\hat{\mathbf{p}}, \mathbf{q})$, this runner-up has a chance to be the final winner and should be inserted in the heap. This procedure is repeated until $l(\hat{\mathbf{p}})$ equals L, that is, $LB^{l(\hat{\mathbf{p}})}(\hat{\mathbf{p}}, \mathbf{q})$ is calculated at level L, which is exactly the desired distance $\|\hat{\mathbf{p}} - \mathbf{q}\|_p$. Since the lower bounds of the distance of other points, in the heap or not, are all larger than $\|\hat{\mathbf{p}} - \mathbf{q}\|_p$, the nearest neighbor $\hat{\mathbf{p}}$ is obtained.

The proposed algorithm is summarized below:

```
    /* Preprocessing Stage */
100 given sample points ${\mathbf{p}_i | i = 1, \ldots, s}$ in $R^{2^L}$ space
110 construct $(L+1)$-level structure for each point $\mathbf{p}_i$
120 sort all the sample points according to $\mathbf{p}_i{}^0$
    /* Nearest Neighbor Search Stage */
130 given a query point $\mathbf{q}$
140 begin
150   construct $(L+1)$-level structure for point $\mathbf{q}$
160   find the temporary winner, $\hat{\mathbf{p}}$, using binary search
170   sequentially search for the runner-up, $\tilde{\mathbf{p}}$, from $\hat{\mathbf{p}}^0$
180   $l(\hat{\mathbf{p}}) = 0$
190   calculate $LB^{l(\hat{\mathbf{p}})}(\hat{\mathbf{p}}, \mathbf{q})$
200   $LB(\hat{\mathbf{p}}, \mathbf{q}) := LB^{l(\hat{\mathbf{p}})}(\hat{\mathbf{p}}, \mathbf{q})$
210   insert $\hat{\mathbf{p}}$ into an empty heap
220   while $l(\hat{\mathbf{p}}) < L$ do
230     $l(\hat{\mathbf{p}}) := l(\hat{\mathbf{p}}) + 1$
240     calculate $LB^{l(\hat{\mathbf{p}})}(\hat{\mathbf{p}}, \mathbf{q})$
250     $LB(\hat{\mathbf{p}}, \mathbf{q}) := LB^{l(\hat{\mathbf{p}})}(\hat{\mathbf{p}}, \mathbf{q})$
260     DownHeap($\hat{\mathbf{p}}$)
270     choose the top element in the heap as $\hat{\mathbf{p}}$
280     if $\|\tilde{\mathbf{p}}^0 - \mathbf{q}^0\|_p < LB(\hat{\mathbf{p}}, \mathbf{q})$
290       add $\tilde{\mathbf{p}}$ into the heap and set $\hat{\mathbf{p}} = \tilde{\mathbf{p}}$
300       sequentially search for the new runner-up $\tilde{\mathbf{p}}$
310     endif
320   endwhile
330   output $\hat{\mathbf{p}}$
340 end
```

2.3. Extension to k-nearest neighbors search

The proposed algorithm can be easily extended to find k nearest neighbors, $k > 1$, in the following way. Once the nearest neighbor $\hat{\mathbf{p}}$ is obtained by using the algorithm described in Section 2.2, we can delete it from the heap and continue the process until the second nearest neighbor is obtained. By repeating the above procedure, one can obtain the third nearest neighbor, the fourth nearest neighbor, ..., and so on, until all the k nearest neighbors are obtained. The following pseudo-code can be added to the basic algorithm, in the appropriate order as designated, to provide k nearest neighbors:

```
215 for $i = 1, 2, \ldots, k$
331   delete $\hat{\mathbf{p}}$ from the heap
332   if heap is empty
333     add $\tilde{\mathbf{p}}$ into the heap and set $\hat{\mathbf{p}} = \tilde{\mathbf{p}}$
334     sequentially search for the new runner-up $\tilde{\mathbf{p}}$
335   endif
336 endfor
```

2.4. Points within a distance threshold

In many pattern recognition applications, a query object is recognized with high confidence only when it is sufficiently close to an object in the sample set. Therefore, the distance between the nearest point and the query point should be smaller than a pre-specified distance threshold ϵ. The proposed algorithm can be easily modified to meet this requirement and to further speed up the search process by adding the following pseudo-code:

```
225 if $LB(\hat{\mathbf{p}}, \mathbf{q}) > \epsilon$ stop
325 if $LB(\hat{\mathbf{p}}, \mathbf{q}) > \epsilon$ stop
```

When all the points within the distance threshold ϵ are required, the additional pseudocode for providing k nearest neighbors, given in Section 2.3, can be further added (in this case, let $k = s$).

2.5. Points close to the nearest point

In some cases, all the points that are sufficiently close to the nearest neighbor may be considered as good matches to the query point. To achieve this goal, all the points of distance smaller than $(1 + r)\|\hat{\mathbf{p}} - \mathbf{q}\|_p$ have to be found out, where $\hat{\mathbf{p}}$ is the nearest neighbor. Our algorithm can be easily modified to provide this functionality. After the nearest neighbor $\hat{\mathbf{p}}$ and the minimum distance $\|\hat{\mathbf{p}} - \mathbf{q}\|_p$ are obtained, the threshold ϵ is set to $(1 + r)\|\hat{\mathbf{p}} - \mathbf{q}\|_p$ and the method described in Section 2.4 can be used to provide all the points having distance smaller than ϵ.

3. Experimental Results

To analyze the the performance of the winner-update algorithm, we perform some experiments for three factors, including the size s of the sample set, the dimension d of the feature space, and the distance between the query point and its nearest neighbor. In these experiments, l_2 norm is used as the distance measure and the goal is to find the nearest neighbor. The exhaustive search algorithm and the winner-update algorithm are implemented on a Sun Ultra-1 workstation and the execution time is compared.

We randomly generate the sample set with a pre-specified size s and dimension d. For each sample point, the value for each dimension is randomly generated from a uniform distribution with extent $[0, 1)$. Randomly choosing a point

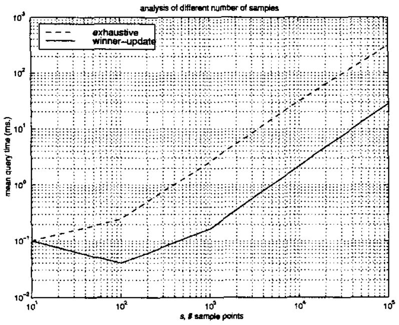

Figure 1. Comparison of query time for s.

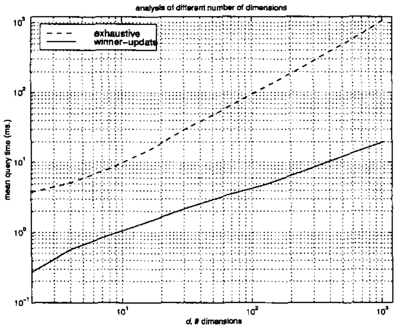

Figure 2. Comparison of query time for d.

from the sample set, a query point of the same dimension d can be generated by adding uniformly distributed noise, with extent $[-e, e)$, to each component of the sample point. The query point is then used to search for its nearest neighbor in the sample point set and the mean query time is compared. The query time of the winner-update algorithm includes the hierarchical structure construction and the search process.

In the first experiment, five different sizes ($10^i, i = 1, \ldots, 5$) of the sample point set is generated. Each point has a dimension of 32 and the noise extent e is set to 0.01. Figure 1 shows the comparison of the mean query time by using exhaustive search algorithm and the winner-update algorithm. When s is small, the computation time of constructing hierarchical structure cannot be ignored and the computational saving is relatively small. As s increases, the winner-update algorithm is roughly ten times faster than the exhaustive search algorithm. This scale is not related to s.

In the second experiment, sample point sets with ten different dimension, $d = 2^l$, $l = 1, \ldots, 10$, are generated. Each set contains 10000 points and the noise extent e is set to 0.01. When $d = 1024$, as Figure 2 shows, the winner-update algorithm is over 50 times faster than the exhaustive search algorithm. Without the curse of dimensionality that K-dimensional binary search tree algorithm suffers, on the contrary, the computation speed of the winner-update algorithm over the exhaustive algorithm scales up.

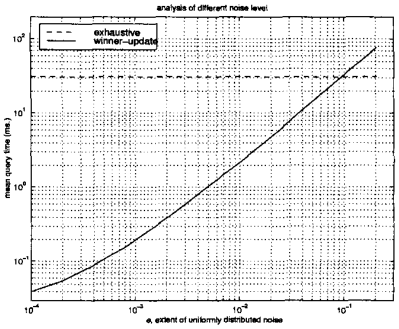

Figure 3. Comparison of query time for e.

The last experiment analyzes how the noise extent e influences the performance of the winner-update algorithm. When e is large, the distance between the query point and its nearest neighbor is large. The computational cost of the winner-update algorithm increases because more samples can have similar distances from the nearest neighbor. We generate the sample point set containing 10000 points with dimension 32. Figure 3 shows that the performance with different noise extent. Notice that when e equals 0.1, the computational cost of the proposed algorithm equals to that of the exhaustive algorithm.

4. Conclusions

In this paper, we have proposed a fast algorithm, called the winner-update algorithm, for nearest neighbor search. This algorithm can reduce a large amount of computation, while need only twice the memory storage and moderate preprocessing. When the distance between the query point and its nearest neighbor is relatively small, the proposed algorithm is particularly efficient. Moreover, the proposed algorithm can be easily modified to provide k nearest neighbors, neighbors within a specified distance threshold, and neighbors close to the nearest neighbor.

References

[1] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[2] K. Fukunaga and P. M. Narendra. A branch and bound algorithm for computing k-nearest neighbors. *IEEE Trans. on Computers*, 24:750–753, 1975.

[3] C.-H. Lee and L.-H. Chen. A fast search algorithm for vector quantization using mean pyramids of codewords. *IEEE Trans. on Communications*, 43(2/3/4):1697–1702, 1995.

[4] S. A. Nene and S. K. Nayar. A simple algorithm for nearest neighbor search in high dimensions. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 19(9):989–1003, 1997.