Applying Multiple KD-Trees in High Dimensional Nearest Neighbor Searching

Shwu-Huey Yen, Chao-Yu Shih, Tai-Kuang Li, Hsiao-Wei Chang

***Abstract*—**Feature matching plays a key role in many image processing applications. To be robust and distinctive, feature vectors usually have high dimensions such as in SIFT (Scale Invariant Feature Transform) with dimension 64 or 128. Thus, accurately finding the nearest neighbor of a high-dimension query feature point in the target image becomes essential. The kd- tree is commonly adopted in organizing and indexing high dimensional data. However, in searching nearest neighbor, it needs many backtrackings and tends to make errors when dimension gets higher. In this paper, we propose a multiple kd-trees method to efficiently locate the nearest neighbor for high dimensional feature points. By constructing multiple kd-trees, the nearest neighbor is searched through different hyper-planes and this effectively compensates the deficiency of conventional kd-tree. Comparing to the well known algorithm of best bin first on kd-tree, the experiments showed that our method improves the precision of the nearest neighbor searching problem. When the dimension of data is 64 or 128 (on 2000 simulated data), the average improvement on precision can reach 28% (compared under the same dimension) and 53% (compared under the same number of backtrackings). Finally, we revise the stop criterion in backtracking. According to the preliminary experiments, this revision improves the precision of the proposed method in the searching result.

***Keywords*—**feature matching, nearest neighbor searching (NNS), kd-tree, backtracking, best-bin-first, projection.

I. Introduction

FEATURE matching is very important to many image processing applications. This issue is equivalent to the optimization problem for finding nearest points in metric spaces [1], [2], [5]-[7]. The nearest neighbor search (NNS) problems have been developed in a rich literature. Linear searching, the simplest method, works for small databases but quickly becomes intractable as either the size or the dimensionality of the problem is large. To solve this problem, several space-partitioning methods including kd-tree have been developed [3], [4]. Kd-tree is a kind of binary tree which iteratively bisects the search space into two regions containing half the points of the parent region. Queries are performed via traversal of the tree from the root to a leaf by evaluating the query point at each split [8].

In 2004, Lowe proposed the SIFT (Scale Invariant Feature Transform) to extract and describe feature points in object recognition application [9]. A local descriptor of dimension 64 or 128 is used for feature points. To match feature points in such high dimensions Lowe used a kd-tree to organize the feature points of the database and a backtracking method called best bin first (BBF) with limited number of backtrackings [10]. Kd-tree with BBF performs quite well and has been adopted in many research works. However, since its result is merely an approximate nearest neighbor, improving the accuracy of finding the exact nearest neighbor under the same number of backtrackings becomes very attractive.

In this paper, we propose a multiple kd-trees method to efficiently search the nearest neighbor of high dimensional feature points. The motivation of the proposed algorithm is based on the fact that two near points are always close under different projections; but two not-so-close points are still not so close for most of projections and, if there is any, may turn out to be close under one or two particular projections. Thus, by increasing the number of different projections, we can increase the probability of finding the exact nearest neighbor of the query point.

The remaining of the paper is organized as following. In Section 2, we give a review on kd-tree and BBF. In Section 3, our method is proposed and the time complexity is analyzed. Experimental results are given in Section 4. Finally our conclusion and future work are stated.

II. Related Work

In this section, we give a review on the nearest neighbor searching methods including kd-tree and the BBF algorithm.

A. The Nearest Neighbor Searching Methods

Though the exact nearest neighbor can be found by brute-force searching, it only works for small databases and quickly becomes impractical as either the size or the dimensionality of the problem becomes large. Thus many improvements have been proposed. Hashing and indexing are among those efforts [11]-[13]. However, it is difficult to find an appropriate hashing function for high dimensional data so that points can be allotted to the hashing table uniformly [14].

Space partition methods are commonly used in the NNS and the kd-tree is one of the most widely used in dealing with high dimensional data [15]-[18]. However, the accuracy is not ideal and gets worse as the dimension gets higher. Backtracking is a way to improve the performance of kd-tree but has a tradeoff of complexity in computation.

B. Kd-Tree with Best Bin First

In 1997, Beis and Lowe proposed an algorithm combining kd-tree and the BBF method (BBF in short) on priority queue

[10]. The kd-tree is built by iteratively bisecting search space on the medium of the dimension with the greatest variance. A query point traverses the constructed kd-tree and the distances to each split (branching point) are recorded on a priority queue. Let *D* be the distance of the query point and the leaf. In backtracking phase, split distances of branching points recorded in the priority queue are compared with *D*. If the distance at a branching point is smaller than *D*, the corresponding not-yet-traversed branch will be traversed. The backtracking is stopped if either the queue is empty or it reaches the allowed maximum backtracking number. The returned nearest neighbor is the one with the minimum distance so far. BBF is designed to efficiently find an approximate nearest neighbor in high dimensional spaces that, according to [10], it returns the exact nearest neighbor for a large fraction of queries and a very close neighbor otherwise.

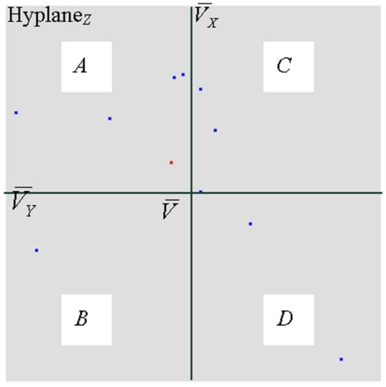
III. The Proposed Method

The kd-tree recursively projects feature points into two lower-dimensional hyper-planes according to the branching conditions. But, false positive may occur after projection. For example, a query point A=(5,5) and two data points B=(7,4), C=(5,10). Clearly the nearest neighbor of A is B because of d(A,B)=√5 and d(A,C)=5. When projecting to X-axis, A'=5, B'=7 and C'=5. The nearest neighbor of A becomes C in the projected space. This error can be resolved by backtracking with a computation burden especially when the dimension of feature point or/and size of data set increases. We propose an algorithm to project data points into three hyper-planes and to build two different kd-trees from each hyper-plane. Finally, the nearest neighbor is the minimum one from these nearest neighbors found from different perspective.

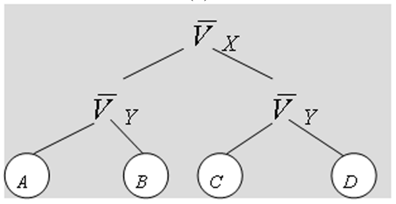
A. Multiple kd-Trees

Assuming *n* is the number of data points, *d* is the dimension of data, V*i* = (v*i*1,v*i*2,…,v*id*) is the *i*th data point *i* =1, 2,…, *n*. Data points are first projected into three hyper-planes to view these data from different perspective. Three axes are located from dimensions which have the largest variances. Without loss of the generality, assume these three axes are X, Y, Z and points are projected to hyper-planes X=0, Y=0, and Z=0. These hyper-planes are denoted by hyplaneX, hyplaneY, hyplaneZ. On each hyper-plane, points are divided into four parts according to the centroid given in (1). For example, assuming data are in 3-dim, in hyplaneZ, the points are divided into four parts according X $=\overline{V}_X$ and Y $=\overline{V}_Y$ as indicated in Fig. 1(a). Note that $\overline{V}_Z = 0$ in (1) for points on the hyplaneZ. The cases for hyplaneX and hyplaneY are similar. Figure 1(b) shows the corresponding tree of Fig. 1(a) with the first split on X= $\overline{V}_X$ and then split on Y = $\overline{V}_Y$.

$$\overline{V} = (\frac{1}{n}\sum_{i=1}^{n} v_{i1}, \frac{1}{n}\sum_{i=1}^{n} v_{i2}, ..., \frac{1}{n}\sum_{i=1}^{n} v_{id}) = (\overline{V}_1, \overline{V}_2, ..., \overline{V}_d) \quad (1)$$



(a)



(b)

Fig. 1 (a) Data points are projected into hyplaneZ and divided by X $=\overline{V}_X$ and Y $=\overline{V}_Y$. (b) The corresponding tree.

After three corresponding trees are built, kd-trees are constructed on four leaves of the trees. That is, as in Fig. 1(b), kd-trees are constructed from points A, B, C, and D. But, we build kd-trees by two different ways. The first one is the conventional one, i.e., build the kd-tree by iteratively bisecting search space according to the slitting hyper-plane on the medium of the dimension with the greatest variance. The second one is similar to the conventional one except the first split is on the dimension that has the second largest variance. Again, we expect projecting points on different perspectives to provide a better chance of getting the exact nearest neighbor. Fig. 2 (the first kd-tree) and Fig. 3 (the second kd-tree) illustrate the situation where P1, …, P10 are data points and *q* is a query point in a 2-dim plane. The exact nearest neighbor for *q* is P7. As the thick line segments shown in Fig. 2(b), the query point *q* traverses the tree and finally lies in A but the exact nearest neighbor P7 is in B. If we want to find the exact nearest neighbor using this tree we have to take six backtrackings. Fig. 3 shows the second kd-tree that is built with the first split on the medium of the coordinate axis with the second largest variance. As shown in Fig. 3, *q* traverses the tree and reaches the leaf P7 in E which is the exact nearest neighbor of *q*.

To summarize the procedure of building multi-kd trees for the data set, first these data points are projected to three different hyper-planes; then, on each hyper-plane, a binary tree is built in a way that the first two levels are centroid-based and the rest of the levels are kd-trees. Two kinds of kd-trees are constructed, one is the conventional one and the other has the first split on the medium of the axis with the second largest variance. Therefore, there are two binary trees for each projected hyper-plane and a total of six binary trees are formed for the data set. For any given query point *q*, it traverses all these trees with the maximum number of backtrackings allowed just as in the BBF method. It returns the point P to be the nearest neighbor if the distance between q to P is the minimum among all the nearest neighbors found.

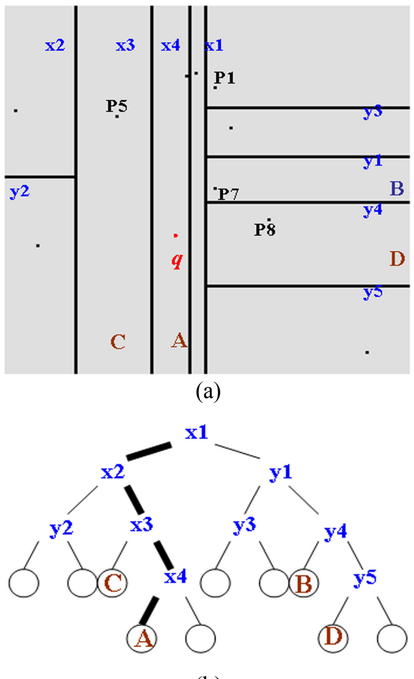


Fig. 2 (a) The conventional kd-tree for points P1, …, P10. (b) The corresponding binary tree where the thick lines constitute the path as q traverses the tree.

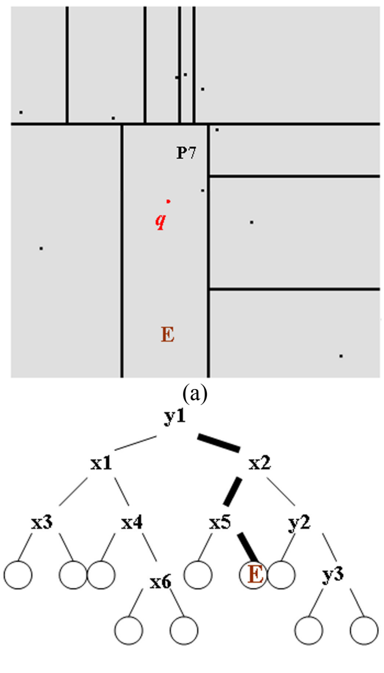


Fig. 3 (a) The kd-tree with the first split on the second largest variance. (b) The corresponding binary tree where the thick lines constitute the path as q traverses the tree.

B. *Complexity analysis*

Assume that n is the number of data and d is the dimension of data. The proposed method needs to compute and sort the variance of each dimension as in conventional kd-tree construction. After projecting points on three hyper-planes with first three largest variances, we use (1) to compute the centroid. The calculations mentioned above are additional comparing to the building kd-tree in the BBF algorithm. In building the rest of trees, the complexity in our method is equivalent to build 24 (= 3x4x2) kd-trees each with $n/4$ points comparing to only one kd-tree with n points in BBF algorithm. Since n is usually much larger than d, the rest of discussion is under the assumption that the dimension d is a constant. Thus, calculations in both variance and centroid are complexity of O(n), and sorting d variances is a constant time. The complexity of building a kd-tree with n points, $T(n)$, has the recurrence relation shown in (2) where the term O($n + n\log n$) is for computing the variances and finding the medium. Solving (2) can find $T(n)$ is equal to O($n\log 2n$) which is also the complexity of building tree in BBF algorithm.

$$T(n) = 2T(\frac{n}{2}) + O(n + n\log n) = 2T(\frac{n}{2}) + O(n\log n) \qquad (2)$$

The overall complexity of building trees in our algorithm is O(n) plus the complexity of building 24 kd-trees of $n/4$ points. That is still O($n\log 2n$) with different constants. In practical situation, we expect the time consumption in our algorithm to be less than 6 times of that in BBF algorithm. For building one complete tree as in Fig. 1(b) (including 4 kd-trees for points in A, B, C, D) is the same of building one conventional kd-tree with less computation required since $\overline{V}_X$ and $\overline{V}_Y$ (from the centroid) are known already. And in total there are six such trees to be built. The experiments later also confirm this claim.

Assume that the maximum number of backtrackings allowed is k and the average length of the backtracking path is half of the tree height. In the proposed method, the maximum backtracking number $k/6$ since the query point traverses six trees. Equations (3) and (4) are the querying complexity of BBF and the proposed method.

BBF:

$$\log n + (1/2\log n) \times k = (1 + k/2)\log n \qquad (3)$$

Proposed:

$$[\log n + (1/2\log(n/4) \times k/6] \times 6 \qquad (4)$$
$$= (6 + k/2)\log n - k$$

where the first term (log n) in (3) and (4) is first time traversing, 1/2 log n in (3) and 1/2 log (n/4) in (4) are the average height when backtracking. There is almost no difference between (3) and (4). The complexity favors ours when k is large.

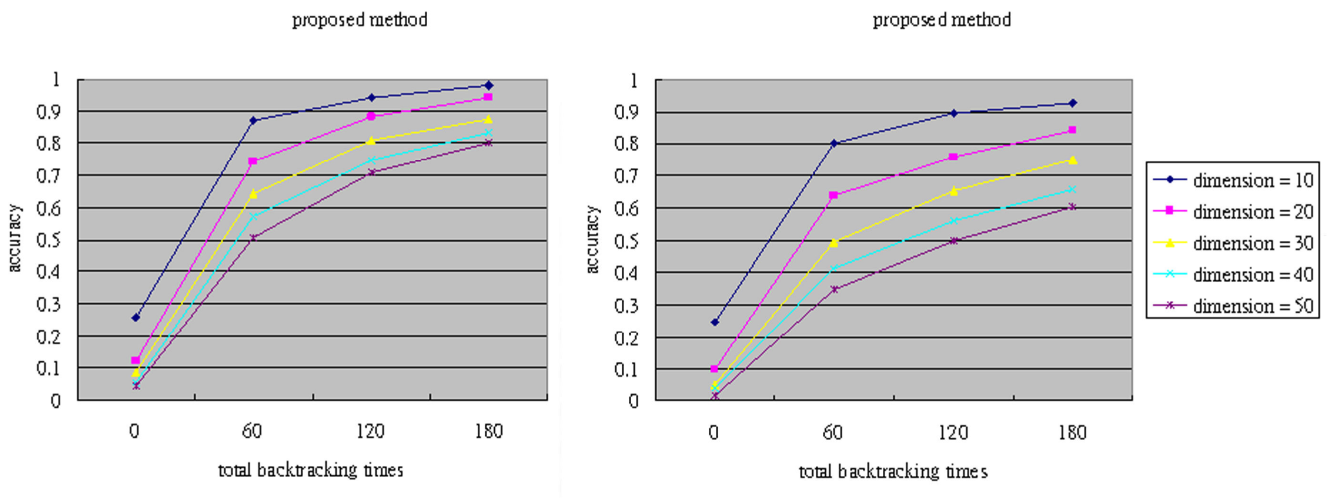
IV. Experiments

The experiments are conducted using Borland C++ Builder 6.0 in an environment of WINDOWS XP SP3, Pentium4 CPU and 512 RAM. The input points and query point are randomly generated real numbers within 0 and 1000. All figures shown in Fig. 4, Fig. 5, and Tables are the average of 1000 repeated tests. The experiments are first conducted in two respects, accuracy and time consumption, of the NNS problem. Finally, a preliminary experiment on stop criteria of backtracking is also performed. Comparisons are made between the proposed method and BBF algorithm.

A. Precision comparison

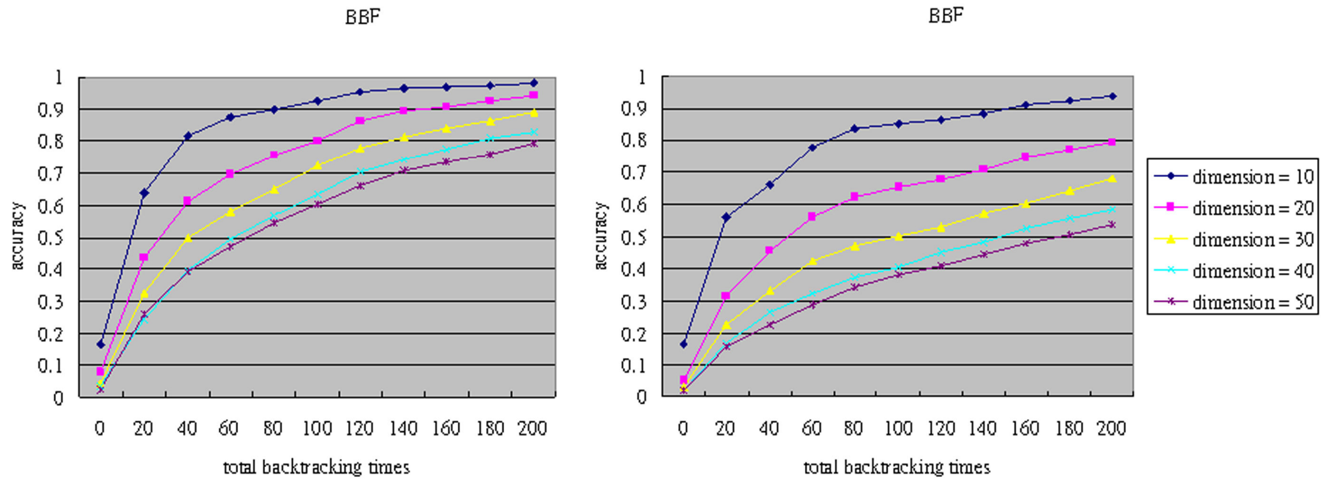
Fig. 4 is a summarization of the accuracy of the proposed method under different data dimension *d* and allowed backtracking number *k*. As in Fig. 4(a), the database size is n = 500, if the dimension of the data is d =10 then the precision is 0.873 with 10 backtrackings on each tree (i.e., k=60), and rises to 0.98 with 30 backtrackings on each tree (i.e., k=180). However, the performance drops if dimension increases. As in d = 50, the precision is 0.505 with k=60 and 0.803 with k=180. Comparing Fig. 4(b) to 4(a), a 4-folds of data size (n = 2000), all the precisions descend. For example, when d = 10 (blue lines), 0.873 drops to 0.803 (k=60), 0.98 drops to 0.925 (k=180); for d = 50 (purple lines), 0.505 drops to 0.347 (k=60), 0.803 drops to 0.604 (k=180). These results confirm that as the dimension d or size of database n increase the precision reduces, and when the number of backtrackings k increases the precision increases.

Fig. 5 is a summarization of the precision results of BBF method under different dimensionalities and the different number of backtrackings. It gives the similar conclusion as in Fig. 4.



(a) n=500 (b) n=2000

Fig. 4 The experimental results of the proposed method under different number of backtrackings



(a) n = 500 (b) n = 2000

Fig. 5 The experimental results of the BBF method under different number of backtrackings

Table I. Precision comparison of two algorithms

(a) n =500 data

# of the back-tracking	Dimension of each point					
	d=10		*d*=20		*d*=30	
	BBF	Ours	BBF	Ours	BBF	Ours
k = 0	0.164	0.256(+0.56)	0.077	0.120(+0.56)	0.043	0.087(+1.02)
k = 60	0.876	0.873(-0.00)	0.696	0.742(+0.07)	0.581	0.643(+0.11)
k =120	0.955	0.941(-0.01)	0.863	0.883(+0.02)	0.779	0.811(+0.04)
k =180	0.974	0.980(+0.01)	0.927	0.942(+0.02)	0.865	0.876(+0.01)
Average	0.742	0.762(+0.03)	0.641	0.672(+0.05)	0.567	0.604(+0.07)
	d=40		*d*=50		Average improved of Ours to BBF	
	BBF	Ours	BBF	Ours		
k = 0	0.035	0.054(+0.54)	0.024	0.041(+0.71)	*k* = 0	+0.679
k = 60	0.493	0.571(+0.16)	0.470	0.505(+0.07)	*k* = 60	+0.080
k =120	0.706	0.747(+0.06)	0.663	0.708(+0.07)	*k* =120	+0.035
k =180	0.809	0.833(+0.03)	0.759	0.803(+0.06)	*k* =180	+0.025
Average	0.511	0.551(+0.08)	0.479	0.514(+0.07)	**+0.205**	

(b) *n* = 2000 data

# of the back-tracking	Dimension of each point					
	d=10		*d*=20		*d*=30	
	BBF	Ours	BBF	Ours	BBF	Ours
k = 0	0.165	0.244(+0.48)	0.051	0.099(+0.94)	0.031	0.051(+0.65)
k = 60	0.780	0.803(+0.03)	0.562	0.637(+0.13)	0.423	0.495(+0.17)
k =120	0.864	0.895(+0.04)	0.677	0.758(+0.12)	0.530	0.652(+0.23)
k =180	0.924	0.925(+0.00)	0.770	0.842(+0.09)	0.643	0.751(+0.17)
Average	0.683	0.717(+0.05)	0.515	0.584(+0.13)	0.407	0.487(+0.20)
	d=40		*d*=50		Average improved of Ours to BBF	
	BBF	Ours	BBF	Ours		
k = 0	0.021	0.037(+0.76)	0.019	0.017(-0.11)	*k* = 0	+0.544
k = 60	0.323	0.412(+0.28)	0.289	0.347(+0.20)	*k* = 60	+0.162
k =120	0.450	0.559(+0.24)	0.410	0.498(+0.21)	*k* =120	+0.169
k =180	0.557	0.657(+0.18)	0.507	0.604(+0.19)	*k* =180	+0.127
Average	0.338	0.416(+0.23)	0.306	0.367(+0.20)	**+0.250**	

Table II. Comparison of two algorithms in high dimension (*n* = 2000 data)

# of the back-tracking	Dimension of each point				Average improved of Ours to BBF	
	d=64		*d*=128			
	BBF	Ours	BBF	Ours		
k = 0	0.011	0.031(+1.82)	0.004	0.007(+0.75)	*k* = 0	+1.284
k = 60	0.235	0.294(+0.25)	0.141	0.185(+0.31)	*k* = 60	+0.282
k =120	0.330	0.439(+0.33)	0.228	0.298(+0.31)	*k* =120	+0.319
k =180	0.442	0.543(+0.23)	0.312	0.390(+0.25)	*k* =180	+0.239
Average	0.2545	0.3268(+0.284)	0.1713	0.220(+0.284)	**+0.531**	

Table I and Table II list the precisions of these two algorithms (under the same dimensionality and the same total number of backtrackings) where the shaded box indicates a better result and the number inside the parenthesis is the improved ratio of ours comparing to BBF algorithm. In these tables, the last column ("Average improvement of Ours to BBF") shows the improved accuracy ratios under fixed numbers of backtrackings, and the last row ("Average") show the improved accuracy ratios under the fixed number of dimensionality. We can observe that in general our proposed method outperforms the BBF algorithm. In Table I, comparing (a) and (b), the proposed algorithm outperforms the BBF more when dimension is higher and the data size gets larger.

As in SIFT, the dimensionality of the feature point can be 64 or 128, and the size of feature points usually is in hundreds to thousands, we simulate a very complex case with data size to be 2000 and dimensionality is 64 or 128 as summarized in Table II. The improvement is consistent and up to 28% for both d in 64 or 128. Observing the results on k = 0, the effectiveness of projecting data into different perspectives in finding the nearest neighbor is also confirmed. Among eleven out of twelve of these tests, our method is better with an average of improvement ratio to be 79.9% (the exception one is when $n = 2000$, $d = 50$, and the ratio would be 72.4% for all 12 tests).

B. Time consumption

Table III is the average time consumption in constructing tree and per backtracking. As analyzed in Section 3.B, the constructing time of the proposed method is approximately six times of that of BBF algorithm. In the experiments, when the dimension d is 50, the tree construction time of ours method is 5.406 times (n = 500) and 4.835 times (n = 1000) of those in BBF method. Overall, the tree construction time is low even in our method (0.62 second for a database size n = 1000 and d = 50). As for the backtracking time consumption, these average numbers from experiments are of total 100, 150, and 200 backtracking. There is not too much difference between these two algorithms; however, our method is a bit less. As in (4) of Section 3.B, when the number of allowed backtracking is large the time consumption will favor our method. And the experiments also show that when the number of allowed backtracking is 200, our time consumption per backtracking is less than half of that of BBF algorithm.

C. Criteria for backtracking termination

When using Best Bin First (BBF), there are two criteria in determining whether backtracking should proceed (as mentioned in Section 2.B). First, the priority queue is empty, i.e., every recorded distance, if there is any, is not smaller than D, the minimum distance of the nearest neighbor of q so far. Second, the maximum number allowed for backtracking is reached. However, according to our observation, the backtracking stops mostly due to the second condition even if the exact nearest neighbor has been found. When evaluating the distance between the query q to the splitting hyper-plane, it is a distance on one-dimension, whereas the stop criterion D is a distance on d-dimension. Thus, we expand the one-dimension distance into a hyper-sphere distance of d dimension. That is equivalent to revise the first constraint in stop criteria of backtracking to be D' which is D divided by $\sqrt{d}$ as shown in (5).

$$D' = \beta \cdot \frac{D}{\sqrt{d}}, \qquad (5)$$

where β is a constant, D is the minimum distance so far, and d is the dimension of the data.

We tested the efficacy of the new constraint D' on a database of $n = 500$ and $d = 50$ with different β values (0.75, 1, 1.25, 1.5, 1.75). Average results from a repetition of 1000 tests are summarized on Table IV, the number in the cell is the average number of backtrackings and the number in the parenthesis is the corresponding precision such that the bolded one meaning precision is the same or better than the original one. According to these preliminary tests, the average number of backtracking is not too much different from those of the allowed maximum number. However, in (a), our proposed method, at the beginning of the backtracking, say 10 backtrackings (a total of 60), the precision is improved most for $\beta = 0.75$ (0.538 comparing to 0.505), then in 20 backtrackings (a total of 120), the precision is improved most for $\beta = 1.25$ (0.724 comparing to 0.708), finally in 30 backtrackings (a total of 180), the precision is improved most for $\beta = 1.25$ (0.815 comparing to 0.803). And in (b), the kd-BBF-tree algorithm with only one tree, the best precision all happens on largest β (1.75) except when the maximum allowed backtracking number is 20 (happened on $\beta = 1.25$). We conclude that the revised D' does not benefit to a shorter execution of backtrackings but it improves the precision to our proposed algorithm especially when $\beta = 1.25$.

Table III. Time consumption (in seconds)

(a) n = 500 and d = 50

Method	Average Time in Tree Construction	Average Time Per Backtracking		
		k=100	k=150	k=200
BBF	0.056123	0.000159	0.000187	0.000315
Ours	0.303400	0.000126	0.000128	0.000128

(b) n = 1000 and d = 50

Method	Average Time in Tree Construction	Average Time Per Backtracking		
		k=100	k=150	k=200
BBF	0.128131	0.000141	0.000235	0.000298
Ours	0.619562	0.000124	0.000118	0.000120

Table IV. The results on revised constraint *D*' on backtracking stop criterion (*n*=500, *d*=50)

(a) Our method

OURS	the average number of backtracking & precision		
# of allowed backtracking	10x6	20x6	30x6
β=0.75	57.582 (**0.538**)	108.303 (0.669)	149.091 (0.740)
β=1.00	59.826 (**0.532**)	119.049 (0.706)	177.042 (0.791)
β=1.25	59.973 (**0.524**)	119.904 (**0.724**)	179.802 (**0.815**)
β=1.50	59.982 (**0.512**)	119.958 (**0.719**)	179.931 (**0.813**)
β=1.75	59.982 (**0.507**)	119.958 (**0.710**)	179.94 (**0.807**)
Original precision	0.505	0.708	0.803

(b) The kd-BBF-method

BBF	the average number of backtracking & precision								
# of allowed backtracking	20	40	60	80	100	120	140	160	180
β=0.75	17.690 (0.216)	28.594 (0.276)	33.532 (0.286)	35.254 (0.292)	35.797 (0.293)	35.979 (0.293)	36.024 (0.293)	36.223 (0.293)	36.223 (0.293)
β=1.00	19.729 (0.249)	38.395 (0.333)	54.194 (0.405)	67.056 (0.445)	76.413 (0.464)	82.525 (0.473)	86.199 (0.477)	88.015 (0.480)	88.905 (0.480)
β=1.25	19.966 (**0.259**)	39.821 (0.366)	59.449 (0.442)	78.652 (0.510)	96.923 (0.560)	114.179 (0.598)	130.120 (0.620)	144.544 (0.639)	157.418 (0.650)
β=1.50	19.980 (0.258)	39.960 (0.386)	59.935 (0.466)	79.881 (0.540)	99.787 (0.606)	119.636 (0.642)	139.295 (0.678)	158.719 (0.710)	177.919 (0.738)
β=1.75	19.980 (0.258)	39.960 (0.391)	59.940 (**0.470**)	79.920 (**0.543**)	99.900 (**0.605**)	119.880 (**0.663**)	139.860 (**0.709**)	159.840 (**0.734**)	179.820 (**0.759**)
Original precision	0.259	0.393	0.47	0.543	0.605	0.663	0.709	0.734	0.759

V. Conclusion and Future Work

This paper proposed a method using multiple kd-trees to find the nearest neighbor in high dimensional space. We build six trees by projecting data into different hyper-planes so that these data can be viewed in different perspective. We compared our method to BBF algorithm. Although our tree constructing time is longer than that of BBF algorithm, but on the whole our construction time is acceptable (not more than 0.62 seconds for 1000 data of dimension 50). Under the same number of total allowed backtracking number, our method almost outperformed on every test. For example, when the dimension of data is 64 or 128, the average improvement on precision can reach 28% (dimension fixed) and 53% (number of backtracking fixed). Experimental results illustrated that the proposed algorithm improves the precision especially when dimension is high and size of data set is large. The effectiveness of projecting data into different perspectives to look for the nearest neighbor is also confirmed by the experiments. Under the condition that no backtracking is used, as our method uses six trees and BBF algorithm uses only one tree, in eleven out of twelve tests our method is better with an average of improvement ratio to be 79.9%.

To understand more on the consequence of backtracking, we performed a preliminary experiment on revising the constraint $D^{'}$ such that $D^{'} = \beta \cdot (D/\sqrt{d})$ such that no more backtracking is performed when every recorded distance in priority queue, if there is any, is not smaller than $D^{'}$ where *D* is the minimum distance of the query point *q* to the splitting hyper-plane and *d* is the dimension of the data. Although the results did not favor shorting the execution numbers of backtrackings, the precision of our proposed method is improved. Moreover, these results seem to indicate that, in one tree, the precision has a relation with β in the constraint $\beta \cdot (D/\sqrt{d})$ as well as the allowed backtracking number. For example, β = 0.75 is better if the number of backtracking is not more than 10; β = 1.25 is better when the number of backtracking is between 10 and 30; β = 1.75 is better when the number of backtracking is between 30 and 60, etc. Further study on this issue is necessary to better understand the relation if there is any.

REFERENCES

[1] D. Fuiorea, V. Gui, F. Alexa, and C. Toma, "A new point matching method for image registration," in *Proc. of 6th WSEAS Int. Conference on Computational Intelligence, Man-Machine Systems and Cybernetics*, 2007, pp. 134–138.

[2] I. Dozorets, I.Gath, and H. Shachnai, "Prototype-based approximate nearest neighbor search", *Advances in Signal Processing and Computer Technologies*, WSEAS Press, 2001, pp. 181–186.

[3] R. Nerino, "Invariant features for automatic coarse registration of point-based surfaces," in *Proc. of the 6th WSEAS Int. Conf. on Signal Processing, Computational Geometry & Artificial Vision*, 2006, pp. 10–15.

[4] A. Moghaddamnia, M. G. Gosheh, M. Nuraie, M. A. Mansuri, and D. Han, "Performance evaluation of llr, svm, cgnn and bfgsnn models to evaporation estimation," in *Proc. of the 9th WSEAS Int. Conf. on Signal Processing, Robotics and Automation*, 2010, pp. 108–113.

[5] M. Brown and D. G. Lowe, "Recognising Panoramas," in *Proc. of the 9th Int. Conf. on Computer Vision*, 2003, pp. 1218–1227.

[6] D. Comaniciu, V. Ramesh, and P. Meer, "Kernel-based object tracking," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. 25 , pp. 567–577, 2003.

[7] F. Isgró, and M. Pilu, "A fast and robust image registration method based on an early consensus paradigm," *Pattern Recognition Letters*, vol. 25, pp. 943–954, 2004.

[8] A. Moore, "An introductory tutorial on KD trees," extracted from Ph.D. Thesis- Efficient Memory-based Learning for Robot Control. Technical Report, No. 209, Computer Laboratory, University of Cambridge, 1991.

[9] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *Int J. of Computer Vision*, vol. 2, pp. 91–110, 2004.

[10] J. Beis and D. Lowe, "Shape indexing using approximate nearest-neighbor search in high-dimensional spaces," in *Proc. of the Int. Conf. on Computer Vision and Pattern Recognition*, 1997, pp. 1000–1006.

[11] A. Califano and R. Mohan, "Multidimensional index for recognizing visual shapes," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. 16, pp. 373–392, 1994.

[12] C. Chen, S. Pramanik, Q. Zhu, and G. Qian, "A study of indexing strategies for hybrid data spaces," *ICEIS 2009 LNBIP*, vol. 24, pp. 149–159.

[13] H. J. Wolfson and I. Rigoutsos, "Geometric hashing: an overview," *IEEE Computational Science and Engineering*, vol. 4, pp. 10–21, 1997.

[14] 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*, vol. 19, pp. 989-1003, 1997.

[15] J. H. Friedman, J. Bentley, and R. Finkel, "An algorithm for finding best matches in logarithmic expected time," *ACM Transactions on Mathematical Software*, vol. 3, pp. 209–226, 1977.

[16] J. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, pp. 509–517, 1975.

[17] J. Bentley, "Multidimensional binary search tree used in database application," *IEEE Trans. on Software Engineering*, vol. 4, pp. 333–340, 1979.

[18] C. S. Anan and R. Hartley, "Optimised k-d-trees for fast image descriptor matching," *2008 IEEE Conference on Computer Vision and Pattern Recognition* (CVPR), pp.1–8.

Shwu-Huey Yen is with Department of Computer Science and Information Engineering, Tamkang University, Taipei, Taiwan (e-mail: shyen@cs.tku.edu.tw)

Chao-Yu Shih is now a firmware R&D engineer in ASUS Tek Computer Inc. in Taipei, Taiwan (e-mail: josh_shih@asus.com)

Tai-Kuang Li is with Department of Computer Science and Information Engineering Tamkang University, Taipei, Taiwan (e-mail:695410141@s95.tku.edu.tw)

Hsiao-Wei Chang is with Department of Computer Science and Information Engineering, China University of Science and Technology (e-mail: changhw@cc.cust.edu.tw)