DIGITAL SIGNAL PROCESSING **7,** 163–187 (1997)
ARTICLE NO. SP970291

Fast Vector Quantization Encoding Based on *K*-d Tree Backtracking Search Algorithm

V. Ramasubramanian* and K. K. Paliwal†

*Computer Systems and Communications Group, Tata Institute of Fundamental Research, Homi Bhabha Road, Bombay—400 005, India; and †School of Microelectronic Engineering, Griffith University, Brisbane, Queensland 4111, Australia

Ramasubramanian, V., and Paliwal, K. K., Fast Vector Quantization Encoding Based on *K*-d Tree Backtracking Search Algorithm, *Digital Signal Processing* **7** (1997), 163–187.

In this paper we consider the *K*-d tree-based backtracking search algorithm and study its performance in the context of vector quantization encoding of speech waveform. We discuss the basic algorithm in detail and highlight the features of optimization as observed from theoretical analysis and from the empirical performance of the backtracking search. It is seen that, despite the 2^K dependence of the theoretical average complexity bound for search in *K*-dimensional vector space, the actual average complexity of the backtracking search in vector quantization encoding of speech waveform is excellent. However, we show that the backtracking search has a very high worst-case computational overhead and that this may be unacceptable in many practical applications such as real-time vector quantization encoding, where the worst-case complexity is of considerable importance in addition to average computational complexity. In vector quantization applications where it is of interest to use large values of K for $r \leq 1$ bit/sample, the codebook size is $N \leq 2^K$ and the backtracking search with a performance bound of the order of 2^K offers a poor worst-case performance. We also consider the use of principal component rotation in the context of vector quantization of speech waveform where there is a high degree of correlation across the components of a vector and show that this can improve the efficiency of optimization and reduce both the main search complexity and the overhead complexity of the backtracking search. © 1997 Academic Press

1. INTRODUCTION

1.1 Vector Quantization Encoding

Vector quantization (VQ) encoding is a minimum-distortion quantization of a vector $\mathbf{x} = (x_1, \ldots, x_K)$ (referred to as the *test vector*) using a set of N *K*-dimensional *codevectors* called the *codebook* $\mathbf{C} = \{\mathbf{c}_i\}_{i=1,\ldots,N}$, of size N, under some distance measure $d(\mathbf{x}, \mathbf{c}_i)$. This involves finding the nearest neighbor of $\mathbf{x}$ in $\mathbf{C}$, given by

$$\mathbf{q}(\mathbf{x}) = \mathbf{c}_j : d(\mathbf{x}, \mathbf{c}_j) \leq d(\mathbf{x}, \mathbf{c}_i), \quad i = 1, \ldots, N,$$

where $\mathbf{c}_j$ is the quantized version of $\mathbf{x}$ and the index j of the codevector $\mathbf{c}_j$ is transmitted in binary form as the channel symbol. The channel symbol in its binary form is an index or address to the N-level codebook $\mathbf{C} = \{\mathbf{c}_i\}_{i=1,\ldots,N}$, and requires $R = \log_2 N$ bits in the transmission. R is the rate of the quantizer in bits/vector and for a *K*-dimensional vector; the rate of the quantizer in bits per dimension is $r = R/K$. Thus, for a bit rate of r bits/sample and dimension K, the codebook size $N = 2^{Kr}$. Since the minimum distortion VQ encoding by an "exhaustive" full search requires N *K*-dimensional vector distance (distortion) calculations per vector, the computational complexity of the full-search vector quantizer grows exponentially with the dimension-rate product (Kr).

For a given rate, rate-distortion theory shows that quantizing a vector instead of a scalar at a time offers lower distortion and that the vector quantizer can achieve the rate-distortion performance bound asymptotically with increase in the vector dimension. However, in practice, it has not been possible to exploit the full theoretical performance potential of vector quantization due to the computational complexity of vector quantization encoding which increases exponentially with the dimension-rate product. The encoding complexity, apart from posing a major obstacle in real-time encoding, also constitutes the most computation intensive step in itera-

1051-2004/97 $25.00
Copyright © 1997 by Academic Press
All rights of reproduction in any form reserved.

tive vector quantizer design [20]. As a result, the general usage of VQ and real-time implementations have so far been limited to bit rates in the range of 1 to 2 bits/dimension and dimension-rate products of 8–12, achieving performances below the full potential of VQ [16, 17, 22, 25].

In order to utilize the full potential of vector quantization and to render them practically useful in real-time coding applications, the problem of reducing its complexity has assumed significant importance and has been receiving considerable attention in the coding community in the recent years. An important approach to circumvent the complexity problem has been to resort to suboptimal vector quantization by using specially structured codebooks such as tree-structured codebooks [7], multistage quantizers [18], and the gain-shape vector quantizers [30, 31]. These quantizers allow higher-dimensional vector coding for the same complexity and render the vector quantizers practically usable for achieving better performance than the conventional quantizers for a given order of complexity. However, these methods compromise the optimality of the codebook and yield a performance inferior to an optimally designed codebook for the given dimension and bit rate. These are indirect methods for dealing with the complexity of vector quantization encoding which do not yield the performance of an unconstrained vector quantizer, as, for instance, in the case of suboptimal codebook structures which reduce the search complexity at the price of optimality of the codebook for a given bit rate and dimension.

1.2. Fast Nearest-Neighbor Search

A direct way of reducing the vector quantization encoding complexity is to circumvent the exhaustive sequential full search of an optimally designed codebook by using fast nearest-neighbor search algorithms. Nearest-neighbor search consists in finding the closest point to a query (or test) point from among a set of N points in K-dimensional space under some distance measure. The problem of finding the nearest neighbor in a multidimensional space arises in several areas such as pattern classification, nonparametric estimation, and information retrieval from multikey data bases and in image and speech data compression using vector quantization. The computational complexity of nearest-neighbor search is a major problem in these areas, when the size N of the point set to be searched becomes very high. As a result, the problem of developing algorithms for fast nearest-neighbor search has attracted significant attention in these areas.

An important approach toward fast nearest-neighbor search in K dimensions is the use of data structures which facilitate fast search of the codebook which is normally unstructured. In this context, the K-d (K-dimensional) tree structure proposed by Bentley [4] is a powerful structure for organizing the search space so as to facilitate a localized search in several proximity search problems. This multidimensional binary tree structure was originally used by Bentley to carry out fast associative searches on multidimensional data for answering a wide range of information retrieval queries such as intersection query, region search, exact match, partial, and closest match queries from files with multiple key records. Subsequently, Friedman *et al.* [15] used the K-d tree structure for fast nearest-neighbor search using a backtracking search strategy. This work is the first to address the issue of optimizing the tree to minimize the expected search time and propose general prescriptions for optimizing the tree.

The basic K-d tree structure has attracted considerable attention in data base query framework and has been used and studied in several related applications [4, 5]. With respect to nearest-neighbor search, the basic backtracking search algorithm of Friedman *et al.* [15] has been employed in several applications and also analyzed further: Anderson [1] uses the K-d tree as a uniform quad-tree realization in an algorithm for reducing pen plotting time. Baird [3] applies it to fast feature identification in structural shape recognition [3]. Equitz [13] uses it to speed up the K-means type training algorithm for vector quantizer design. Moore [24] uses this algorithm for fast nearest-neighbor search in learning robot control. Murphy and Selkow [23] analyze the efficiency of the K-d tree for nearest-neighbor search in discrete space, i.e., files of fixed-length binary key records, and provide guidelines for determining if the construction of the K-d tree will provide an improvement over the exhaustive full search. Eastman and Zemankova [10] address the problem of optimizing and using the K-d tree for partially specified nearest-neighbor queries in the context of document retrieval application where only some of the keys are specified. Filho [14] addresses the problem of optimizing the tree for partial match queries in data base application. Eastman [9] considers the issue of choosing the optimal depth (or, equivalently, the bucket size) of the tree to minimize the total time of the backtracking search including the cost of processing a bucket which holds the data points.

An alternate paradigm for nearest-neighbor search

using the K-d tree structure is based on optimizing and searching bucket–Voronoi intersections, where the K-d tree structure organizes the search space with an explicit consideration of the Voronoi partition [8, 21, 26–29]. The K-d tree structure has also been used to realize clustering algorithms which are faster than the conventional K-means type of iterative algorithm [11–13, 32]. Wan *et al.* [32] use the K-d tree structure for developing a divisive algorithm for multidimensional data clustering which achieves solutions close to the local optimal solutions of the conventional K-means algorithm. This work considers a partitioning scheme which minimizes the sum of squared errors in a clustering context and compares it with two variations of the optimization of Friedman *et al.* [15]. Equitz [11–13] uses the optimization prescribed by Friedman *et al.* [15] to organize the training data into buckets for performing fast agglomerative clustering with emphasis on exploiting the partitioning and space localization offered by the K-d tree in the form of buckets to merge points into clusters with reduced complexity.

In this paper we consider the optimization and backtracking search algorithm proposed by Friedman *et al.* [15] and study its performance in the context of vector quantization encoding of speech waveform. We discuss the basic algorithm in detail and highlight the features of optimization as observed from theoretical analysis and from the empirical performance of the backtracking search. It is seen that, despite the 2^K dependence of the theoretical average complexity bound, the actual average complexity of the backtracking search in vector quantization encoding of speech waveform is excellent. However, we show that the backtracking search has a very high worst-case computational overhead and that this may be unacceptable in many practical applications such as real-time vector quantization encoding, where the worst case complexity is of considerable importance in addition to average computational complexity. In vector quantization applications where it is of interest to use large values of K for $r \leq 1$ bit/sample, the codebook size is $N \leq 2^K$ and the backtracking search with a performance bound of the order of 2^K offers a poor worst-case performance. We also consider the use of principal component rotation in the context of vector quantization of speech waveform where there is a high degree of correlation across the components of a vector and show that this can improve the efficiency of optimization and reduce both the main search complexity and the overhead complexity of the backtracking search. This paper reports the results obtained in an earlier study [26] which was mainly motivated in providing a performance baseline for comparison with an alternate paradigm of using the K-d tree structure in terms of bucket–Voronoi intersections [8, 21, 26–29].

2. THE K-DIMENSIONAL TREE

The K-d tree is a multidimensional generalization of the simple one-dimensional binary tree normally used for searches such as point location, insertion in an ordered list, interval search, and sorting. Primary to the concept of fast search using a binary tree is the idea of halving the search domain by dividing an ordered list (or the original search interval) by means of a simple scalar comparison against some fixed partition value. A single partition generates two sublists, the left and right. Values less than the partition value form the left list and those with larger values form the right list. Each of these lists is recursively subdivided to generate a tree structure [19]. In the general case of K dimensions, the $\mathcal{R}^K$ space is split into two half-spaces by means of a hyperplane orthogonal to one of the K coordinate axes. Such a hyperplane $\mathbf{H}$, given by $\mathbf{H} = \{\mathbf{x} \in \mathcal{R}^K\colon x_j = h\}$, defines two half-spaces, R_L and R_R, $R_L = \{\mathbf{x} \in \mathcal{R}^K\colon x_j \leq h\}$ and $R_R = \{\mathbf{x} \in \mathcal{R}^K\colon x_j > h\}$. This partitioning hyperplane is represented by two scalar quantities: (i) j—the index of the coordinate axis orthogonal to the plane also referred to as the partitioning or discriminator axis, and (ii) h—the location of the plane on this axis. Any vector point $\mathbf{x}$ can now be located with respect to the dividing plane $\mathbf{H}$ by a single scalar comparison of the form $x_j \leq h$, i.e., the vector's jth component value with the partition value h. The initial region corresponds to the root of the tree at layer 1 and the two subregions R_L and R_R obtained by the division correspond to the left and right sons at layer 2. Each of these two half-spaces is successively divided by hyperplanes orthogonal to the coordinate axes and d such successive divisions starting with the initial region as the root at layer 1 creates a tree of depth d with 2^d terminal regions termed "buckets" at the $(d + 1)$th layer. Clearly, the final partitioning is decided by the choice of the partitioning planes at the nonterminal nodes of the tree. Every nonterminal node is associated with a region and a partitioning hyperplane of the form $\mathbf{x}\colon x_j = h$, which needs storage of just two scalar quantities (j, h) at each node. Given any vector in $\mathcal{R}^K$, a sequence of d scalar comparisons of the vector's jth component value with the partitioning hyperplane (j, h) at that node leads to the leaf (or bucket) containing the vector. Thus a K-d tree structure of depth d partitions the $\mathcal{R}^K$ space into 2^d

disjoint rectangular regions (buckets) and allows identification of the bucket containing a given vector **x** in d scalar comparisons.

The basic structure of the K-d tree is illustrated in Fig. 1 for a planar case. The root region ABCD is divided by hyperplane EF into two halves. Fig. 1b shows the corresponding tree of depth 3 generated by divisions of these two regions by various hyperplanes orthogonal to one of the coordinate axes x_1 or x_2. A vector $\mathbf{x} = \{x_1, x_2\}$ is located to be in the bucket region IEQR after 3 scalar comparisons: x_1 with EF, x_2 with IJ, and x_1 with QR. Similarly, the path corresponding to a vector located in bucket region LKHF will consist of a comparison sequence with the hyperplanes EF, GH, and KL.

The optimization of the K-d tree during its design is a very important issue which decides the efficiency of the tree in reducing the search complexity. The optimization of the tree involves the choice of the dividing hyperplane (the partitioning axis j and the location of the hyperplane h) for each nonterminal node in the tree such that the resulting tree structure when used for nearest-neighbor search yields the minimum search time complexity for a specified search procedure.

In the original tree proposed by Bentley [4], the partitioning axis for each node is chosen on the basis of its level in the tree; i.e., all nodes on any given level of the tree have the same partitioning axis. Starting from the root node with the partitioning axis as $j = 1$, the discriminator for the successive layers are obtained by cycling through the coordinates in order from 1 to K. Subsequently, Friedman *et al.* [15] first used the K-d tree structure for fast nearest-neighbor search using a backtracking search strategy and also provided general prescriptions for optimizing the tree to minimize the expected search time. The algorithm based on this optimization and backtracking search has been shown to have a $O(\log N)$ average complexity performance. In the following sections, we consider this algorithm in detail and study its performance in the context of vector quantization of speech waveform vectors.

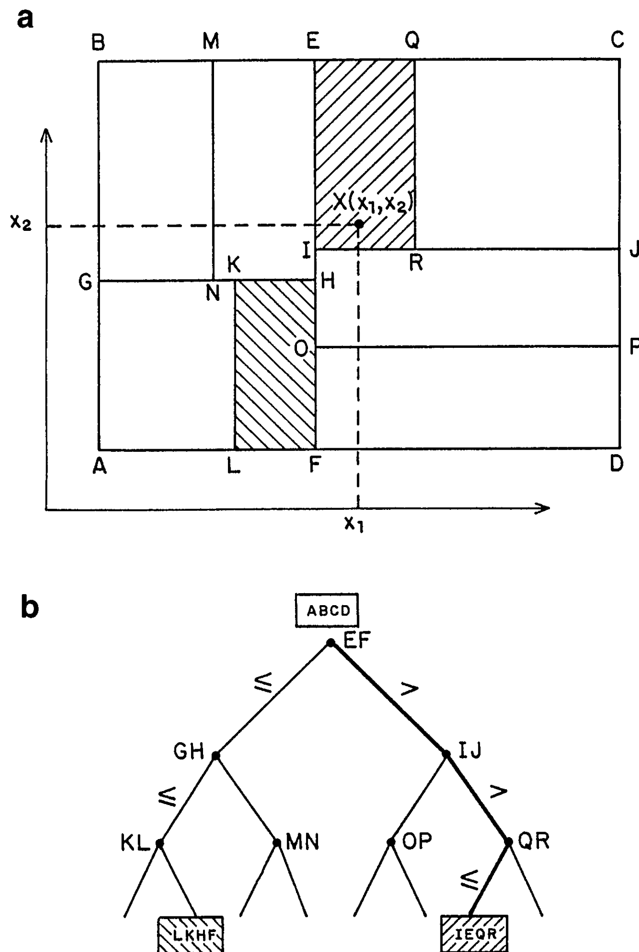


FIG. 1. Example of K-d tree partition.

3. FRIEDMAN–BENTLEY–FINKEL OPTIMIZATION AND BACKTRACKING SEARCH PROCEDURE

3.1. Optimization Criteria

Here, the optimization and backtracking search procedure proposed by Friedman *et al.* [15] is described briefly. This optimization, referred to henceforth as the FBF optimization (for Friedman–Bentley–Finkel), was developed under the restriction that no knowledge of the test data distribution is available and that the only information available is the multiple key records in the file (or data base) which is equivalent to the codevectors in the case of vector quantization. The prescription for the choice of the partitioning hyperplane at every nonterminal node of the tree considers the codevectors lying within the region represented by the node to be partitioned and constitutes a local optimization. The direct application of the FBF optimization for the case of vector quantization encoding will be as follows: (i) the axis along which the corresponding codevector components have the maximum variance is chosen as the discriminant axis and (ii) the median of the corresponding codevector component distribution on the chosen axis is chosen as the partition value. The variance and median are computed using codevectors within the region to be divided. The optimization results in a balanced binary tree, with each bucket containing an equal number of codevectors.

3.2. Backtracking Search Principle

The general nature of the search consists in first finding a tentative (current) nearest neighbor of the given test vector **x** from among the small set of codevectors within the bucket containing **x** and then determining the actual nearest neighbor from among other buckets which overlap with the current nearest-neighbor ball by means of a *backtracking* search procedure. Here, the backtracking search is described in some detail to provide sufficient framework for a subsequent study of the features related to its poor worst-case efficiency and high computational overheads.

Given a test vector **x**, the bucket containing the test vector is first identified. After a search within the small number of codevectors contained in the bucket, the nearest neighbor of **x** in this set and the corresponding nearest-neighbor ball centered at **x** with radius equal to the distance between **x** and the current nearest neighbor are determined. Since it is of interest to ensure optimality of the nearest-neighbor solution with respect to the entire set of codevectors, it has to be verified that no other codevector is closer to **x** than the current nearest neighbor, i.e., no other codevector lies inside the current nearest-neighbor ball. This is satisfied when, after a complete search within the bucket, the nearest-neighbor ball is entirely within the bounds of the bucket region. The search can then be terminated. This test is termed the *ball-within-bounds* (BWB) test in [15]. If this test fails, i.e., if the nearest-neighbor ball is not within the bounds of the bucket region, then the nearest-neighbor ball overlaps with one or more neighboring buckets and a search has to be carried out in these overlapping buckets for a possible update of the current nearest neighbor and the nearest-neighbor ball. Further search then consists in identifying the buckets overlapping the nearest-neighbor ball and in conducting a search in these buckets. Since the buckets are disjoint regions, any other bucket overlapping with the ball will not contain the vector **x** and will lie at the end of a different path in the tree. This can therefore be accessed only by backtracking to the nearest nonterminal node containing the bucket. Since a bucket is a subregion of all its ancestor regions in the path leading to the bucket, the overlap of the nearest-neighbor ball with the bucket manifests in the overlap with all its ancestor regions. This fact forms the basic principle behind the backtracking search which uses a test termed the *bounds-overlap-ball* (BOB) test to identify the regions and the buckets overlapping with the current nearest-neighbor ball.

Thus, the backtracking search consists of two types of overhead operations—the BWB test and the BOB test—which guide the search from the initial bucket containing the test vector to the final bucket containing the actual nearest neighbor until termination of search within the smallest parent node containing the nearest-neighbor ball. The BWB test is a relatively inexpensive computation, consisting of scalar comparisons of the bounds of a hypercuboidal region with the corresponding bounds of the smallest hypercube inscribing the current nearest-neighbor ball. In comparison, the BOB test is computationally more expensive, as this is essentially a vector distance computation. Though it can be realized as a partial accumulated distance [15], the overhead complexity of the backtracking search is dominated by the BOB test for two reasons: (i) it is computed at every node visited during the search, and (ii) the partial accumulated distance computation can incur the full-distance computation cost, since its relative savings over the full-distance cost is highly dependent on the test-vector distribution.

3.3. Recursive Description of the Backtracking Search

The above search is carried out conveniently by a recursive procedure. Here this search is described up to two layers from the terminal layer to bring out the general nature of the recursion. (The basic flow of recursion is illustrated with reference to Fig. 1.) The algorithm is invoked first with the root region (ABCD in Fig. 1) as the region to be searched. The initial current nearest-neighbor distance is set to a large value. If the region is nonterminal, the algorithm performs a test to decide the next successive region to be searched by comparing the appropriate vector component with the partition associated with the node (partition EF vs x_1). The decision identifies one of the two sons representing the region which is on the same side of the partition as the test vector **x**. This amounts to descending to the son containing (or closer to) the vector **x** (region FECD in Fig. 1). The algorithm is then invoked with this region as the search region. If the region is terminal, the codevectors inside the region are searched and the current nearest-neighbor ball is updated if a codevector nearer to **x** than the current nearest neighbor is found. (In Fig. 1, the search reaches the terminal node IEQR containing **x** by this sequence of tests.) The algorithm checks for ball-within-bounds of this bucket. If the test succeeds, i.e., the current nearest-neighbor ball is within the bounds of the bucket, the search is terminated. In case of failure, this indicates an overlap of the ball with other buckets and control returns to the immediate parent node where a

bounds-overlap-ball test is carried out to determine whether the current nearest-neighbor ball overlaps with the bucket opposite to the current partition (i.e., region RQCJ, opposite to partition QR). In case of an overlap, the algorithm is invoked with this opposite region, so as to search the opposite bucket (or farther son) for possible update of the current nearest-neighbor ball. A ball-within-bounds test is done to check termination before control returns to the parent node. In case of no overlap (or if both the sons have been examined), the control returns and the parent node (corresponding to region IECJ) becomes the current node with one of its subtree having been examined completely. The algorithm checks for ball-within-bounds of this node and the search is terminated, if the test succeeds. In case of failure, this indicates an overlap with adjacent regions at this layer and control returns to consider the opposite region (region FIJD) by the bounds-overlap-ball test. In case of an overlap, the algorithm is invoked with the opposite region (region FIJD), so as to examine the *subtree* of this region in which one or more buckets overlap with the current nearest-neighbor ball.

The search thus proceeds by identifying and searching the overlapping buckets one at a time in the order of increasing distance in the tree from the initial bucket, while updating the current nearest neighbor whenever a closer neighbor is found. The recursive procedure which carries out the overall search implicitly performs a backtracking to move from one overlapping bucket to another, the overlap being detected by the bounds-overlap-ball test. The termination is checked by a ball-within-bounds test at the root of every subtree that has been examined completely to ensure that the nearest-neighbor ball is contained within a nonterminal region in which each of the buckets has either been searched or ignored after verifying that the nearest-neighbor ball does not overlap with it.

3.4. Iterative Description of the Backtracking Search

The backtracking search is now illustrated with a simple example. Since the recursive structure of the procedure does not conveniently reveal the serial flow of the search, we first represent the backtracking procedure in an iterative format to bring out the backtracking steps explicitly. The following describes the backtracking search subsequent to the identification of and search inside the bucket containing the test vector **x**. The current nearest neighbor and the current nearest-neighbor ball are thus available and the current node is the bucket containing the test vector **x**.

Backtrack Search

BWB test: Do until **BWB**(*current nearest-neighbor ball, current-node*)

BOB test: *if* **BOB**(*current nearest-neighbor ball, farther-son*)

then current-node = farther-son

examine sub-tree(*current-node*)

else current-node = parent-node [**backtrack**]

endif

enddo

The call "examine sub-tree(*current-node*)" descends to the son representing the region on the *same* side of the current partition as the test vector **x** until the terminal (bucket) node is reached. Then the current node is set to this terminal bucket region, all the codevectors in this bucket are searched, and the current nearest neighbor and the current nearest-neighbor distance are updated if a codevector closer to **x** than the current nearest neighbor is found. The following is a simple recursive representation of "examine sub-tree(*node*)":

examine sub-tree(*node*)

if node is terminal

then search codevectors in bucket; update current nearest-neighbor ball if necessary; return.

else current-node = closer-son; examine sub-tree(*current-node*)

endif

In the "backtrack search" procedure, **BWB**(*current nearest-neighbor ball, current-node*) checks if the current nearest-neighbor ball is within the bounds of the current node and **BOB**(*current nearest-neighbor ball, farther-son*) checks if the current nearest-neighbor ball overlaps with the bounds of the farther-son region; "farther-son" refers to the node opposite to the current partition and "closer-son" refers to the son representing the region on the same side of the current partition as the test vector **x**. The current partition is the partition corresponding to the current node. Explicit details of partition and node boundary parameters are not given for the sake of clarity. It should also be noted that an exact iterative equivalent of the recursion requires maintaining flags to tag the search status of nodes; these are not taken into account in the above iterative description for the sake of simplicity. For instance, it is assumed that the BOB test is done on the farther-son only if it has not been examined already. If the farther-son is

examined, then both sons have been examined and the procedure will backtrack to the parent-node as when the BOB test fails on the farther-son.

3.5. Best- and Worst-Case Situations in Backtracking Search

We make use of the above simple iterative form to trace a typical backtrack search and highlight the intrinsic poor worst-case efficiency of the backtracking search and its overhead complexity in terms of the BWB and BOB tests. The exact flow of the search as obtained from the iterative program above can be serialized as a sequence of the following main steps:

1. Examine sub-tree.
2. Descend to closer-son and search bucket.
3. Compute distance with codevector in the bucket.
4. Update current nearest neighbor, if necessary.
5. Perform BWB test in current node.
6. Perform BOB test on the farther-son node.

This is shown in Fig. 2, for search within an example partitioning by the FBF optimization criterion for a set of eight points in 2-dimensional space shown in Fig. 3. Shown alongside is the corresponding tree structure of depth $d = \log N = 3$, where the nonterminal nodes are labeled by the partitioning planes and the terminal nodes by the bucket regions. For ease of description, the tree is also shown with all the nodes labeled from *a*–*o* and each leaf (bucket) tagged with the codevector it contains. The search, when the test vector $\mathbf{x}$ is located within the bucket GHKL (node *i*) as shown in Figs. 4 and 5, is described by the search sequence under "case (i)" and "case (ii)" in Fig. 2, respectively.

In case (i), (corresponding to Fig. 4), the search first descends to the bucket GHKL (node *i*) and computes the distance between $\mathbf{x}$ and the codevector $\mathbf{c}_4$ inside this bucket. Since the current nearest-neighbor ball is completely within the bounds of the bucket GHKL, the search terminates. This is a best-case search situation where just one bucket is examined and only one distance is computed.

On the other hand, if the test vector is located in the bucket GHKL as shown in Fig. 5, the backtracking search has a very poor efficiency and suffers from very high computational overheads. It can be noted that the current nearest-neighbor ball (after the computation of the distance between $\mathbf{x}$ and $\mathbf{c}_4$) overlaps with all the buckets and will therefore give rise to a search within all the buckets. The complete search sequence for this situation is shown under "case (ii)" in Fig. 2. First, on failure of the BWB test with the bucket node *i*, a BOB test is conducted on the farther-son node *h*. Since this test succeeds, node *h* is searched; however, since $\mathbf{c}_6$ is outside the current nearest-neighbor ball, no nearest-neighbor update is done. A BWB test in node *h* fails and the search backtracks to parent-node *d* where the BWB test fails. The search then considers the farther-son node *e* with a BOB test, which succeeds. The sub-tree of node *e* is then examined, by first descending to the closer-son node *k* and then searching this bucket. This is followed by a BWB test on node *k* and a BOB test on node *j* which results in the search of bucket *j*. The search then backtracks to the parent-node *e* and then to node *b* after which the search continues into the other sub-tree (node *c*) of the root node. The search examines this entire sub-tree with two current nearest-neighbor updates ($\mathbf{c}_5$ and $\mathbf{c}_7$), searches all buckets except node *m*, and terminates after a BWB test in the root node *a* with $\mathbf{c}_7$ as the actual nearest neighbor. The search by then has searched 7 buckets, carrying out 7 distance computations, 7 BOB tests, and 14 BWB tests for searching a set of 8 codevectors. This is a very poor efficiency situation and represents a typical worst-case behavior of the backtracking search.

4. PERFORMANCE CHARACTERISTICS OF BACKTRACKING SEARCH

4.1. Theoretical Characteristics

Under the backtracking search strategy, the expected search time minimization was formulated in [15] in the form of minimizing the average number of buckets which overlap with the current nearest-neighbor ball. The exact analysis of such an optimization formulation was intrinsically difficult due to the limitation that the search was of a backtracking nature in addition to the restriction that the test vector distribution is unknown. This rendered the performance analysis obtuse and indirect. The analysis therefore did not directly yield the main prescriptions for the optimal division of a region and these were provided by means of qualitative considerations. The maximum variance criterion for the choice of axis is used so that (i) the geometric shape of the buckets will be reasonably compact, and (ii) the probability of the partition intersecting with the nearest-neighbor ball (corresponding to a test vector lying on either side of the partition) will be small, thus reducing the probability of the current nearest-neighbor ball overlapping with an adjacent bucket.

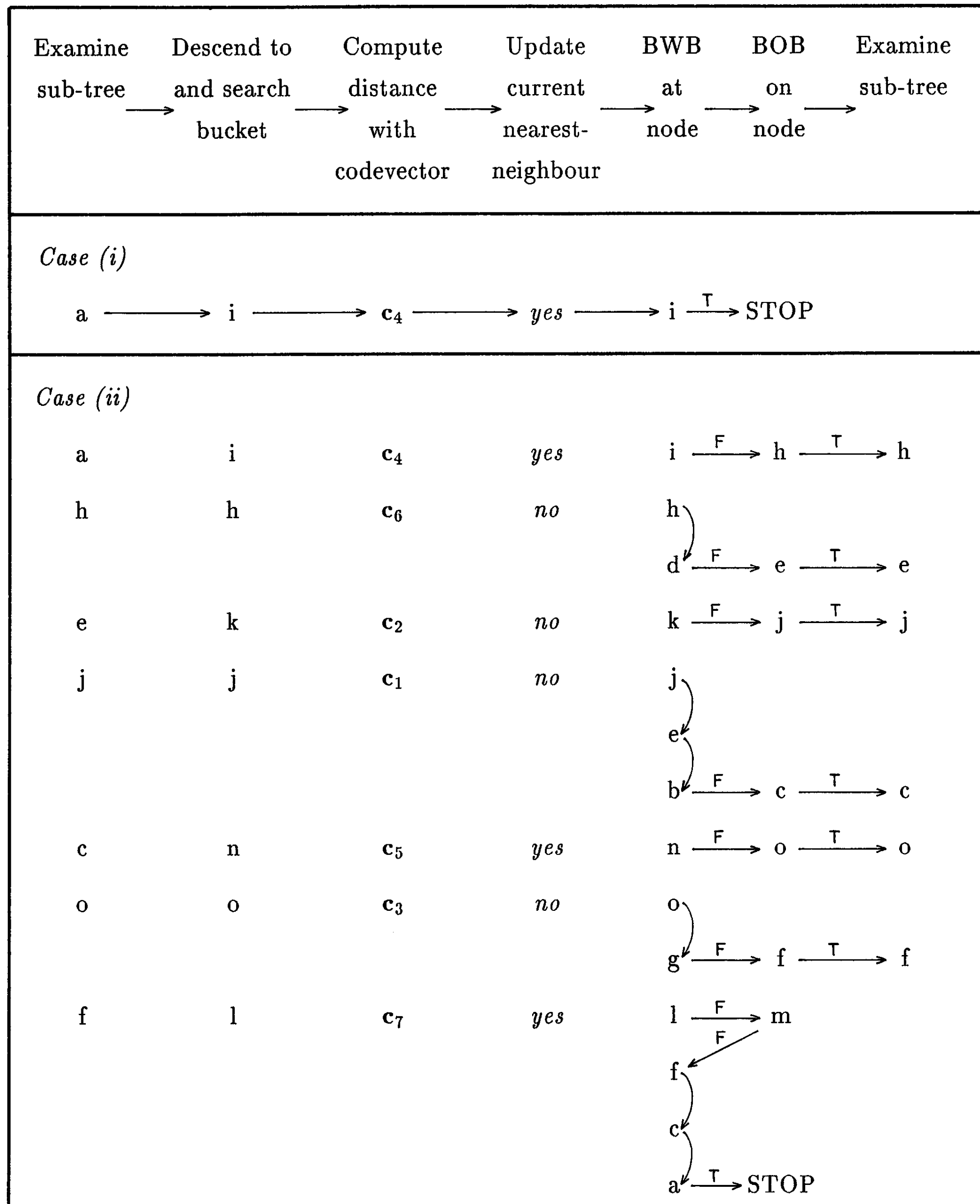
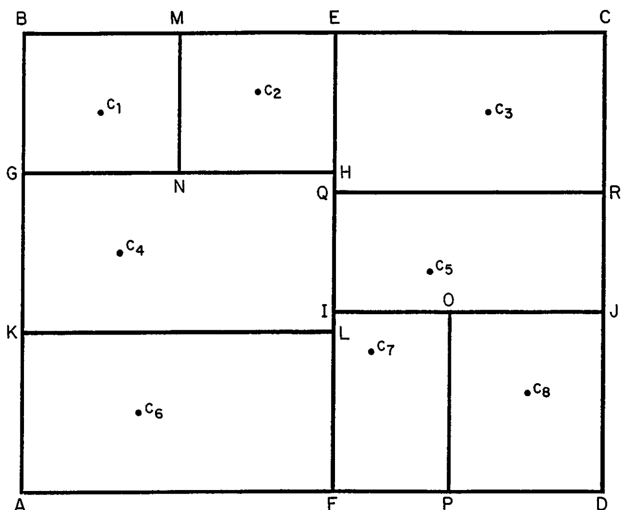
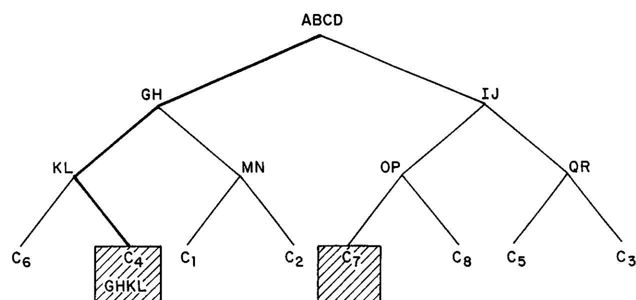


FIG. 2. Backtracking search serial flow.

The motivation behind the choice of median as the partition value on the chosen axis is to have an equally likely outcome resulting in a maximum information binary division.

The analysis, however, helps in finding the optimal value for the number of codevectors in a bucket region (which is related to the depth of the tree) and the corresponding upper bound for the average number of codevectors checked. The upper bound for the average number of buckets checked is obtained as the average number of buckets overlapped by the smallest hypercube that contains the actual nearest-neighbor ball. This is obtained in [15] as $\{[(m/b)\,G(K)]^{1/K} + 1\}^K$, where b is the number of codevectors contained in a bucket, m is the number of nearest neighbors sought, and $G(K)$ is a geometric constant equal to the ratio of a unit hypercube to the volume of a constant unit radius ball in K-dimensional space. $G(K)$ is a function of dimension K and the distance measure used in defining the constant unit radius ball. The upper bound $b\{[(m/b)\,G(K)]^{1/K} + 1\}^K$ for the expected number of codevectors checked is minimized with respect to b to yield an optimal value of $b = 1$ for the bucket size. Correspondingly, the





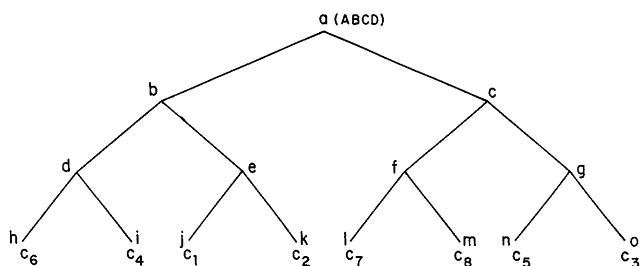


FIG. 3. Example of K-d tree partition by FBF optimization for 8 points in 2 dimensions; tree depth = 3; number of buckets = 8; codevectors/bucket = 1.

upper bound of the average number of codevectors checked is $\{[mG(K)]^{1/K} + 1\}^K$. Significantly, this upper bound is independent of the codebook size and the test vector distribution. While the above upper bound directly determines the extent of backtracking to be done for a given dimension, the constancy of the number of codevectors searched with codebook size and the fact that the time taken to descend the tree is logarithmic with the number of nodes in the tree together imply a search time logarithmic in the codebook size N for log N tree depth.

Since $G(K) \geq 1$ for the L_p distance measure, the

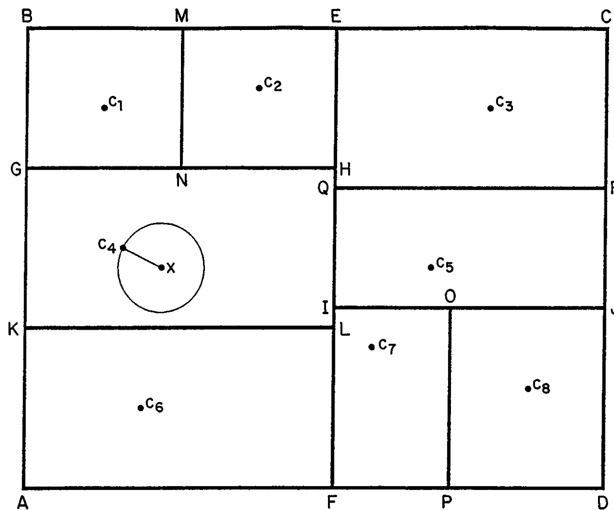


FIG. 4. Best-case situation for backtracking search.

upper bound has an exponential (2^K) dependence on the dimensionality. For the L_∞ norm the smallest hypercube containing the nearest-neighbor ball is the ball itself and therefore $G(K) = 1$. The upper bound, which then is $(m^{1/K} + 1)^K$, is also the actual average number of codevectors checked. For the nearest-neighbor case where $m = 1$, the actual average number of codevectors checked is therefore 2^K. Since the number of buckets overlapped by a ball of constant volume decreases with increasing p, the $p = \infty$ (L_∞) result serves as a *lower bound* for all vector space p-norms. Therefore, the actual average search complexity of the backtracking search, as obtained by the analysis has a lower bound of 2^K. This is also observed in the simulation results reported by Friedman *et al.* [15]. In this context, it is

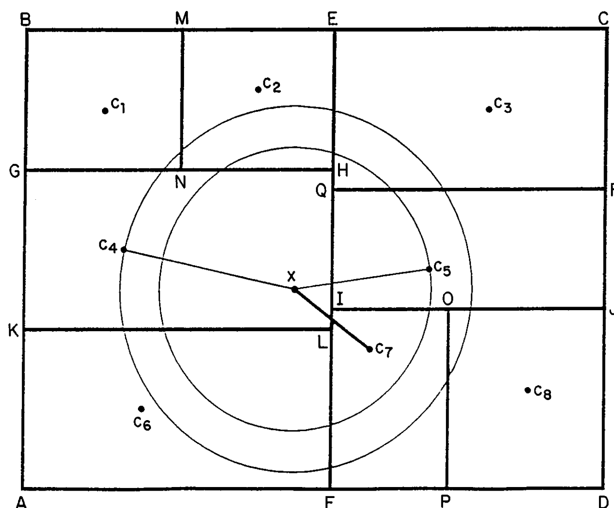


FIG. 5. Poor-efficiency situation for backtracking search.

interesting to relate this to the exponential dependence of the constant of search with dimension in the cell-based spiral search method [6]. Both the K-d tree based backtracking algorithm and the cell-based search algorithm share a similar basic search localization principle. As a result, the increase with dimension of the number of buckets overlapping a nearest-neighbor ball in the backtracking search can also be viewed geometrically as due to the fact that the number of neighboring buckets of any given bucket in a K-dimensional space is of an exponential order in K. This also relates to the adjacency of a point or the number of sphere touchings in a K-dimensional space as observed in [6].

4.2. Empirical Characteristics

While the above-mentioned analysis of [15] pertains to the search complexity based on the intrinsic geometric aspect of the number of buckets overlapping the nearest-neighbor ball, the actual backtracking search contributes to the search complexity with high overhead operations which is dominated by the bounds-overlap-ball test computed at every node visited during the search. The bounds-overlap-ball test is essentially a vector distance computation, though actually realizable in a computationally less expensive form as done in [15] based on a partial accumulated distance. In the following, we note some of the main factors contributing to the inefficiency of the backtracking search.

The optimal choice of bucket size, $b = 1$, suggested by the analysis for achieving minimum expected search time in terms of the number of buckets examined does not minimize the total cost which is also determined by other factors such as cost of fetching and processing a bucket and the backtracking overhead costs of performing the ball-within-bounds and bounds-overlap-ball test computations. In order to obtain an efficient trade-off between the backtracking overheads and the actual number of codevectors checked, Friedman *et al.* [15] suggest and also verify empirically that it is computationally more efficient to have larger bucket sizes. Eastman [9] explicitly considers the overhead associated with processing a bucket in obtaining the optimal bucket size which minimizes the total time of the backtracking search. This work analytically validates the observation made in [15] that bucket sizes larger than 1 has to be chosen for achieving minimum overall search complexity in the backtracking algorithm. It also obtains the optimal bucket size as a function of the ratio of the cost of processing a bucket to the cost of examining a point in a bucket in addition to the other factors as obtained originally in [15].

The analysis pertains to a search sequence where the current nearest-neighbor ball is also the actual nearest-neighbor ball. However, in practical cases, the first nearest-neighbor ball is not guaranteed to be the actual nearest-neighbor ball and is consequently larger than that. The amount of overlap and the extent of backtracking needed dictated by the ball is hence larger than for the actual nearest-neighbor ball. The poor performance of the analytically optimal choice of $b = 1$ in comparison to an empirically chosen value could possibly be attributed to this effect also.

The backtracking search examines the overlapping buckets in the order of increasing distance from the initial bucket only with respect to the tree structure. This does not always constitute the optimal order spatially, since there always exist cases where an overlapping bucket containing the actual nearest neighbor, which may be *spatially* adjacent to the current bucket, actually belongs to the opposite subtree of the root node. The search, then, has to backtrack upto to the root node to gain access to this overlapping bucket and suffers an extreme inefficiency. The worst-case search situation shown in Fig. 5 is a typical illustration of this problem.

The empirical performance characteristics of the backtracking search such as the average logarithmic search cost and exponential dependence with dimension, originally shown in [15] has also been independently observed and confirmed in another recent study [24]. Moore [24] uses the backtracking search algorithm for fast nearest-neighbor mapping used in learning of robot control and studies the empirical performance of the algorithm with tree size, dimension and underlying distribution dimensionality. In particular, Moore shows that the search cost increases very quickly with the dimensionality of the distribution. Moore also considers some simple and heuristic alternate partitioning rules to account for the skewness of the data point distribution in generating trees which are balanced but do not have unequally proportioned bucket regions.

5. SIMULATION RESULTS

We now present some results characterizing the performance of the backtracking search in the context of vector quantization encoding of speech waveform. First, we show a typical partitioning produced by the FBF optimization for speech waveform codebooks in 2 dimensions. Figure 6 shows the partition-

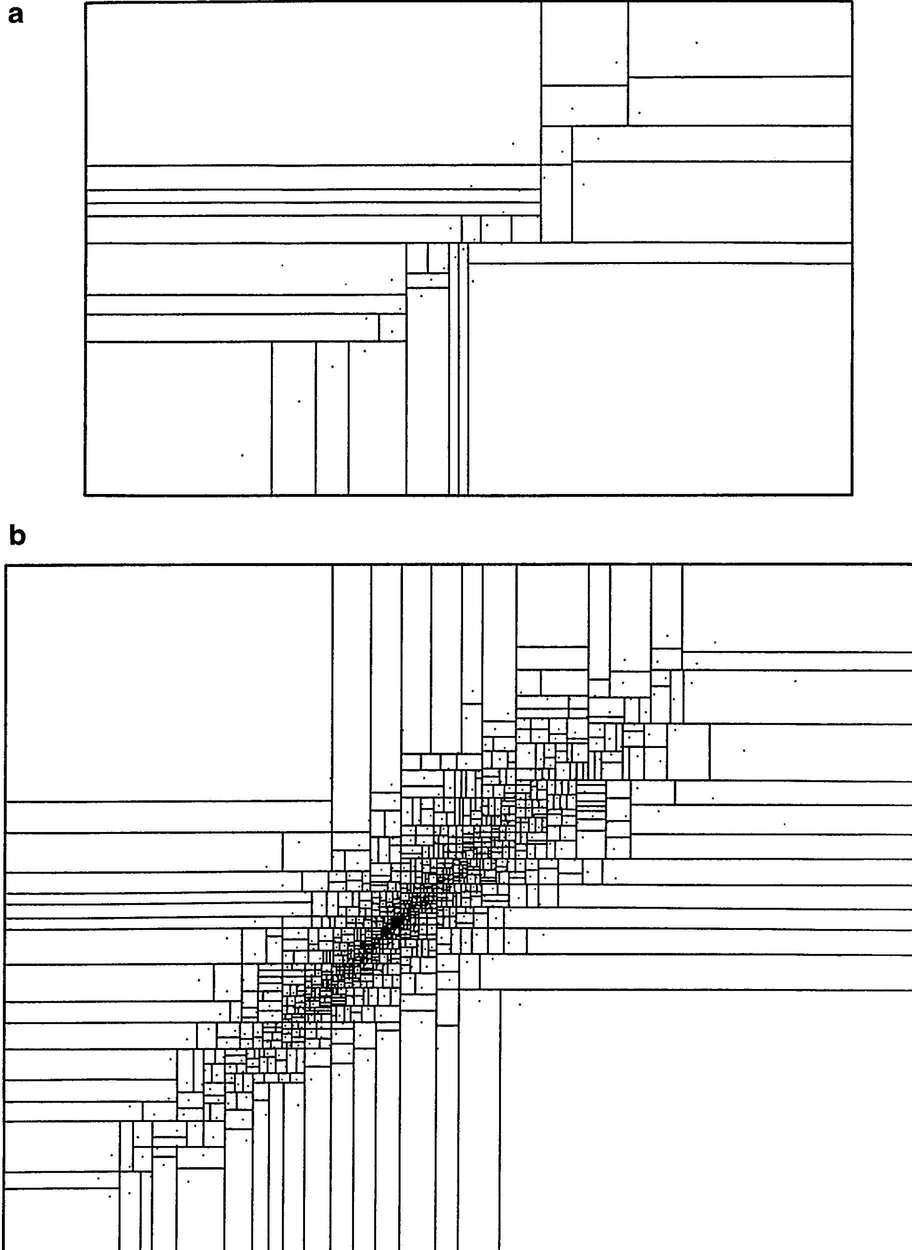


FIG. 6. K-d tree partitioning by FBF optimization of speech waveform codebooks of dimension $K = 2$ and size (a) $N = 32$ and (b) $N = 1024$.

ing of codebooks of size $N = 32$ and 1024 and vector dimension $K = 2$ with tree depth log N, each bucket containing one codevector. The basic complexity measures used here to characterize the performance of the backtracking search are as follows:

1. N_{cc}: Number of codevectors checked, i.e., whose distance to the test vector is computed; this represents the main complexity of the search,
2. N_{bob}: Number of bounds-overlap-ball tests carried out; this represents the main overhead complexity of the search,
3. N'_{bob}: Effective number of bounds-overlap-ball tests when it is implemented in the computationally less expensive partial distance form [15].

As noted earlier, the overhead computations in the backtracking search actually consists of the ball-within-bounds and bounds-overlap-ball tests. Here we consider only the complexity of bounds-overlap-ball test as this is essentially a vector distance computation and is therefore a more dominant computation than the ball-within-bounds test which mainly comprises of comparisons. The bounds-

overlap-ball test complexity can thus be added to N_{cc} to yield a measure of the overall complexity of the backtracking search as $N_{cc} + N'_{bob}$. $N_{cc} + N_{bob}$ is the overall complexity with the full bounds-overlap-ball test and corresponds to the actual complexity in terms of the number of buckets and codevectors examined. The complexity of the ball-within-bounds test as incurred in the algorithm specified in [15] is considered separately.

5.1. Backtracking Search Complexity as a Function of Tree Depth

Figures 7a and 7b show, respectively, the average and worst-case complexity in terms of the measures given above as function of tree depth from 1 to the maximum depth of log $N = 10$ for a codebook of size $N = 1024$ and dimension $K = 8$. This was obtained using 50,000 vectors of speech waveform data (50 s of speech from multiple speakers sampled at 8 kHz). The ordinate of the graph is in log scale to show the main search complexity N_{cc} and the overhead complexity N_{bob} in a same comparative scale and to cover the large variation in the measures over the range of tree depth shown. The following can be noted from this:

1. The basic complexity of number of codevectors checked N_{cc} decreases significantly with increase in tree depth. At depth 10, while the average complexity is excellent (22.7), the worst-case complexity is very high (542); this is only half the full-search cost of 1024.
2. The overhead complexity N_{bob} can be seen to increase with increase in tree depth, reaching values comparable to N_{cc}. For tree depths approaching log N (= 10, here), N_{bob} can even be higher than N_{cc}. For small tree depths, the N_{bob} value is small since the bucket sizes are large and the extent of backtracking is less. As tree depth increases, the extent of backtracking increases sharply. In particular, the worst-case complexity of N_{bob} increases from 1 for tree depth 1 to 675 for depth 10.
3. The partial distance realization of the bounds-overlap-ball test with complexity N'_{bob} is significantly less than the full bounds-overlap-ball test complexity N_{bob}; this reduces the overhead complexity of N_{bob} by more than a factor of 2.
4. The overall complexity of the search $N_{cc} + N_{bob}$ decreases initially with tree depth and increases again, conforming to the basic analysis that the theoretically optimal bucket size of 1 does not result in the empirically best performance and that the bucket size should be chosen to optimally trade-off the decrease in the basic complexity N_{cc} and the increase in the backtracking overhead complexity N_{bob} with tree depth. The worst-case overall complexity $N_{cc} + N_{bob}$ can even be larger than the full-search complexity at higher tree depths (i.e., small bucket-sizes), for instance, being 1217 for tree depth 10.
5. $N_{cc} + N'_{bob}$ is however considerably smaller than $N_{cc} + N_{bob}$. The average overall complexity is very low, being 33.6 at tree depth 10. However, the worst case complexity is very high, having its lowest value of 740 at tree depth 9 (bucket size of 2).

We now show an interesting measure that directly characterizes the extent of backtracking done during a typical search. In Fig. 8, we show the probability that the search has to backtrack up to a particular depth in the tree. This result is obtained for dimension $K = 8$, codebook size $N = 1024$ using a tree of depth 10 on 50,000 vectors of speech waveform data. This measure can be explained as follows: Starting from the bucket layer the search backtracks to examine other buckets. As seen earlier, in order to reach any other bucket, the search has to backtrack to the nearest nonterminal node whose sub-tree contains the bucket. When the actual nearest neighbor is found, the search can terminate only at a nonterminal node where the ball-within-bounds test is satisfied. This is the lowermost layer in the tree which the search has to reach in its entire backtracking. Figure 8 essentially shows the probability that the search is terminated at a depth d, with $1 \leq d \leq 11$ in this case. Thus, if the search is terminated by a search within the bucket that contains the test vector, there is no backtracking and this adds to the frequency count for $d = 11$ (corresponding to the leaf layer) in obtaining the histogram shown. If the search backtracks up to the root node corresponding to $d = 1$, this represents an inefficient search situation, since this implies that half the buckets in the tree have already been considered (either searched or ignored after a bounds-overlap-ball test failure) and that the search proceeded to the other half in the opposite sub-tree of the root node. From the probability histogram shown, it can be seen that the probability of backtracking to the root node at depth $d = 1$ is very high (about 0.4) and decreases drastically to a very low value of 0.003 for the leaf layer at depth $d = 11$. This clearly demonstrates the very high overhead complexity of the backtracking search and the inefficiency incurred in having to examine spatially adjacent buckets lying on the other half of the tree at the root node.

In summary, we have shown the relative magnitudes and behavior of the basic search complexity and the overhead complexity of the backtracking search as a function of tree depth. We have observed the typical trade-off involved in these two complexi-

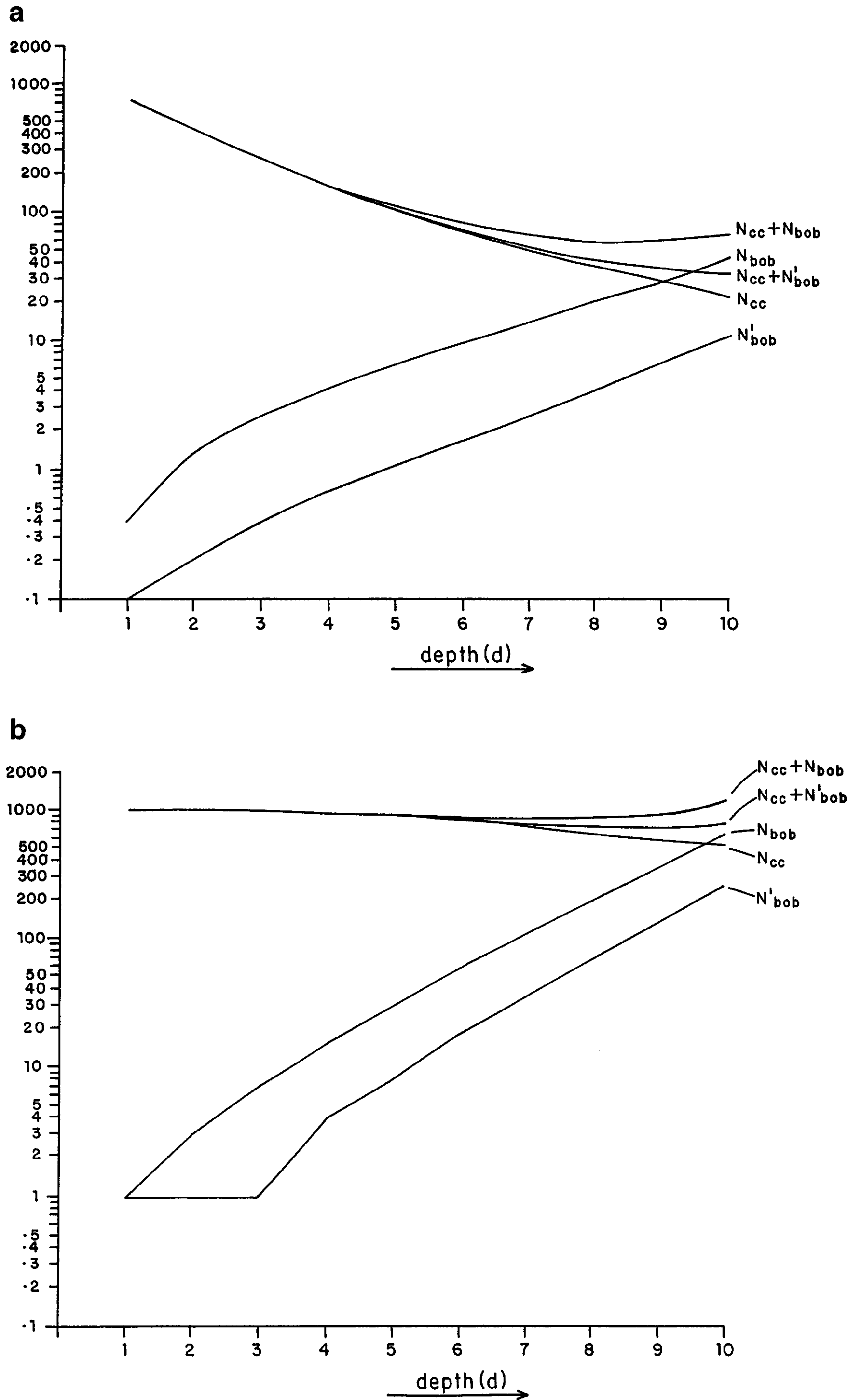


FIG. 7. (a) Average and (b) worst-case complexities of backtracking search for tree-depth 1 to 10 for codebook size $N = 1024$ and dimension $K = 8$.

ties which results in minimum overall complexity for depths less than log N with bucket size larger than the theoretically optimal value of 1. The very high overall worst-case complexity and its relatively insignificant decrease (in comparison to the average complexity) with tree depth are also clearly brought out. It is also shown that the worst-case inefficiency is intrinsic to the backtracking search as it backtracks to the root with a very high probability and has a very low probability of termination at the leaf layer.

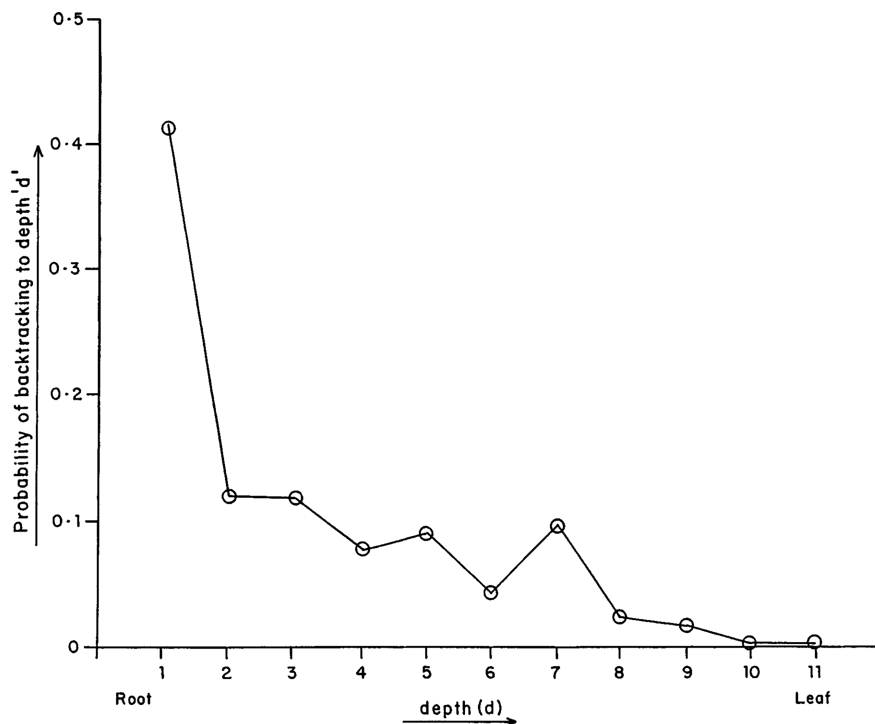


FIG. 8. Extent of backtracking. Dimension $K = 8$ and codebook size $N = 1024$.

5.2. Backtracking Search Complexity with Codebook Size

In Fig. 9, we show the complexity reduction performance of the backtracking search for codebook sizes $N = 32, 64, 128, 256, 12$, and 1024 and vector dimension $K = 8$, for tree depth log N. Figures 9a and 9b show, respectively, the average and worst-case complexity of the measures described above. These were obtained using 50,000 vectors of speech waveform data as the test vectors. The relative magnitudes of N_{cc}, N_{bob}, and N'_{bob} and the overall complexities $N_{cc} + N_{bob}$ and $N_{cc} + N'_{bob}$ can be noted. The backtracking search can be seen to have an excellent average complexity performance, but a very poor worst-case complexity performance, with very high $N_{cc} + N'_{bob}$ values. The worst-case complexity of $N_{cc} + N_{bob}$ that will be incurred without the use of the partial distance realization of the bounds-overlap-ball test can be seen to be higher than the full-search complexity for all the codebook sizes. However, the worst-case $N_{cc} + N'_{bob}$ is significantly less, but still very close to the full-search complexity.

5.3. Backtracking Search Complexity with Dimension

In Fig. 10, we show the complexity reduction performance of the backtracking search for various dimensions $K = 2, 4, 6, 8$, and 10, and fixed codebook size $N = 1024$ for tree depth log $N = 10$. Figures 10a and 10b show, respectively, the average and worst-case complexity of the measures described above. The same observations about the relative magnitudes of the various measures as in the case of varying codebook sizes can be made here. In addition, the effect of increasing dimensionality in drastically increasing the complexity of backtracking search can be noted clearly. In particular, the worst-case complexity can be seen to increase significantly and more drastically with increase in dimension. In these simulations for vector quantization encoding of speech waveforms, the average complexity is significantly very low and interestingly does not have the 2^K lower bound as expected theoretically or as observed in the simulation results of Friedman *et al.* [15] using uniform and normal distributions. However, the drastic worst-case complexity behavior with increasing dimensionality is clearly indicative of the exponential dependence with dimension of the number of buckets overlapping a nearest-neighbor ball.

5.4. Ball-within-Bounds Overhead Complexity

We now consider the overhead complexity due to the BWB test which is used to check the termination

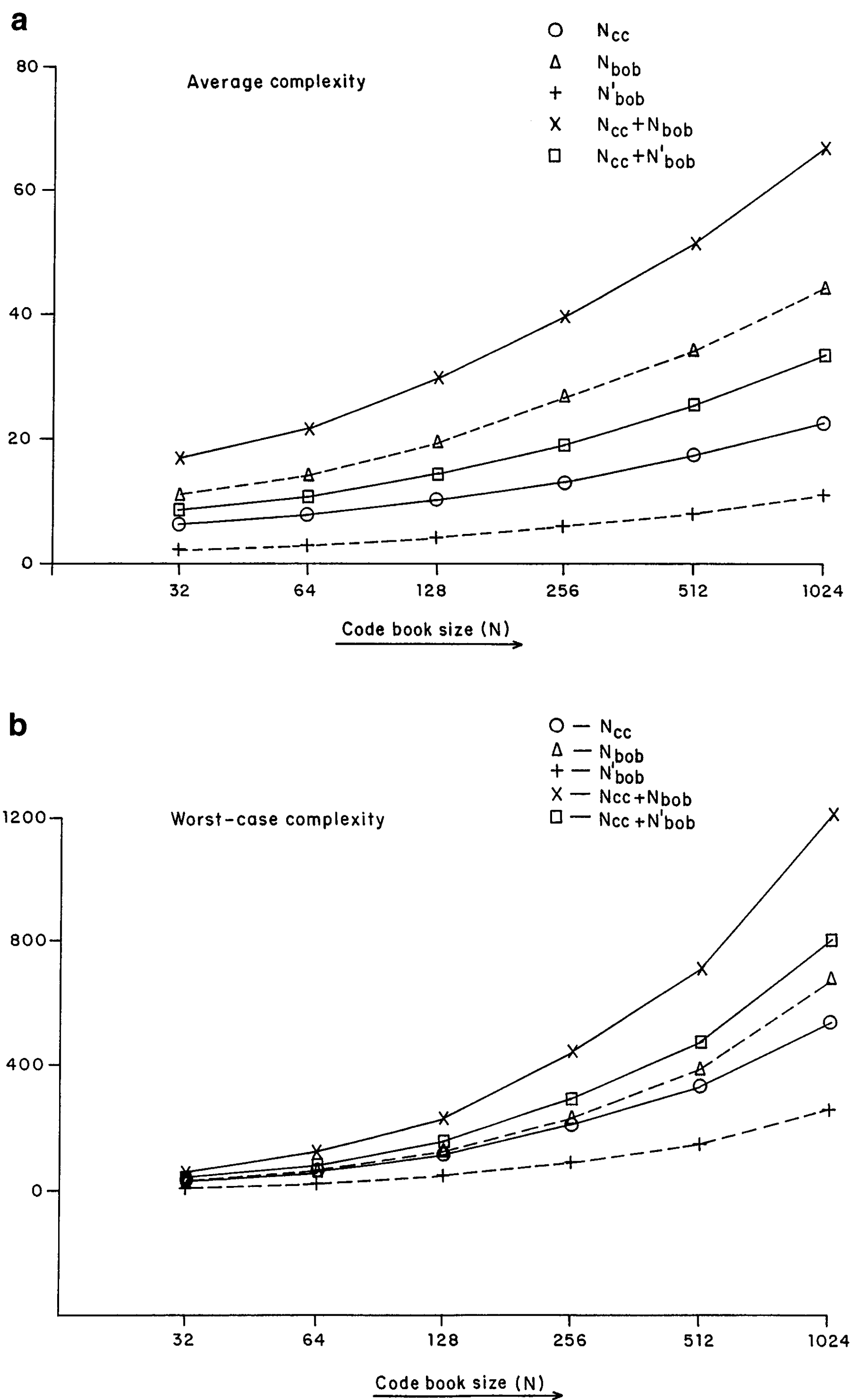


FIG. 9. (a) Average and (b) worst-case overall backtracking search complexities for codebook sizes $N = 32$ to 1024; dimension $K = 8$; tree depth $d = \log N$.

of the search. The recursive realization given in [15] carries out this test at every region (both terminal and nonterminal) that has been examined before returning to the corresponding parent-node. As seen above, since the backtracking overhead complexity is very high, the number of ball-within-bounds computation is very high with a worst-case complexity close to 2^d for a tree of depth d. Thus, though the BWB test consists only of comparisons, its contribution to the overhead complexity of the backtracking search can

be very high in the algorithmic realization given in [15]. We refer to this as F-BWB, for "Friedman *et al.*'s BWB" realization.

5.4.1. Reduced BWB tests. Here we show that the BWB test needs to be carried out only at the nonterminal nodes containing the test vector and that this reduces the maximum number of BWB tests carried out per test vector to $d + 1$. The number of BWB tests is minimum (equal to 1) when the actual nearest neighbor is found inside the bucket containing the test vector. Failing this, the current nearest-neighbor ball as found within this bucket (containing the test vector) is not enclosed within the bucket and the search backtracks to examine other buckets. However, the nearest-neighbor ball cannot be contained within the bounds of the overlapping bucket regions since the center of the nearest-neighbor ball (i.e., the test vector) does not lie in these buckets. Therefore, the search can be terminated by the BWB test, only when the nearest-neighbor ball is within the bounds of a nonterminal node which contains the test vector. Of course, as seen earlier, this nonterminal node is such that its subtree has been completely examined; i.e., each of the buckets within this nonterminal region has been either searched or ignored after verifying that the nearest-neighbor ball does not overlap with it. The nonterminal nodes that contain the test vector are referred to here as the "proper ancestors" of the bucket containing the test vector and the maximum number of such "proper ancestors" is d for a tree of depth d. Therefore, the BWB test needs to be done only at these nonterminal nodes and the search incurs a BWB complexity of atmost $d + 1$ BWB tests (including the test done at the first bucket) in the worst-case situation when the search backtracks upto to the root node. We refer to this as the R-BWB, for "reduced BWB tests."

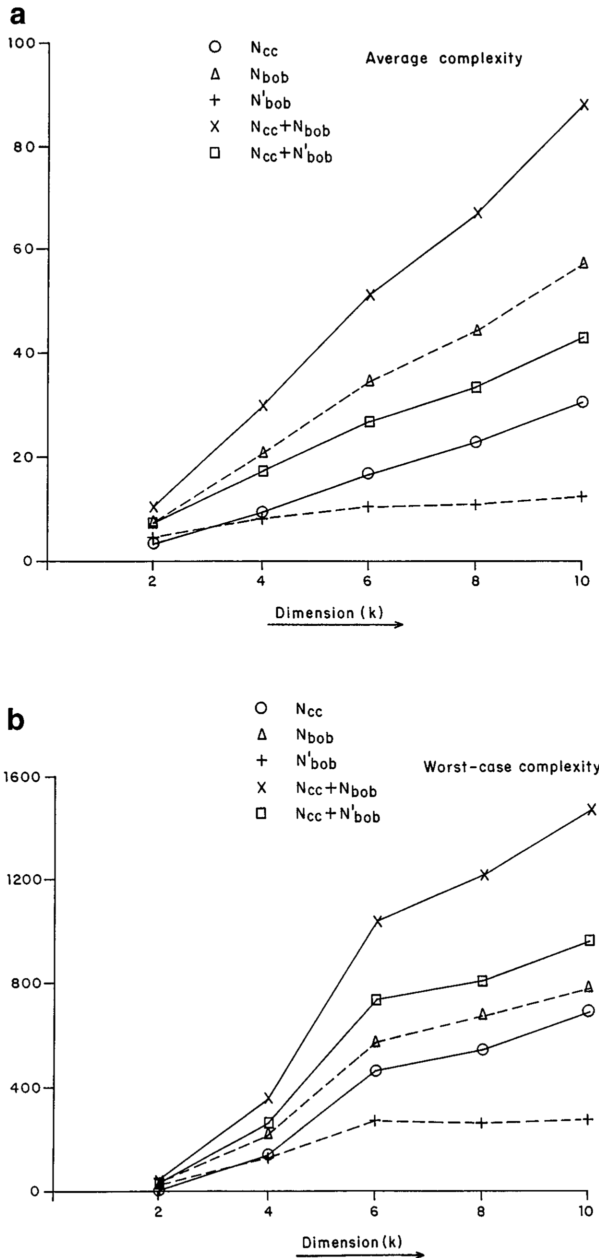


FIG. 10. (a) Average and (b) worst-case overall backtracking search complexities for dimensions $K = 2, 4, 6, 8$, and 10; codebook size $N = 1024$; tree depth = 10.

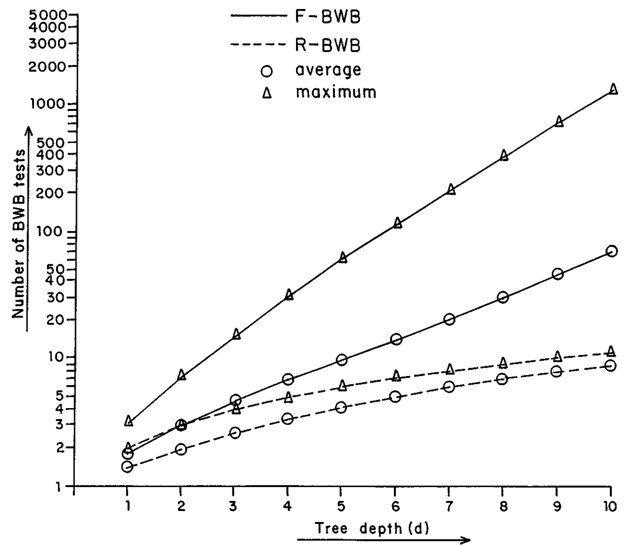


FIG. 11. Ball-within-bounds complexity for tree depth 1 to 10; dimension $K = 8$ and codebook size $N = 1024$.

Figure 11 shows the average and maximum number of F-BWB and R-BWB tests carried out for tree depths 1 to 10 using a codebook of size $N = 1024$ and dimension $K = 8$. It can be clearly seen that the number of F-BWB tests is very high, with its maximum being close to 2^d. In comparison, the number of

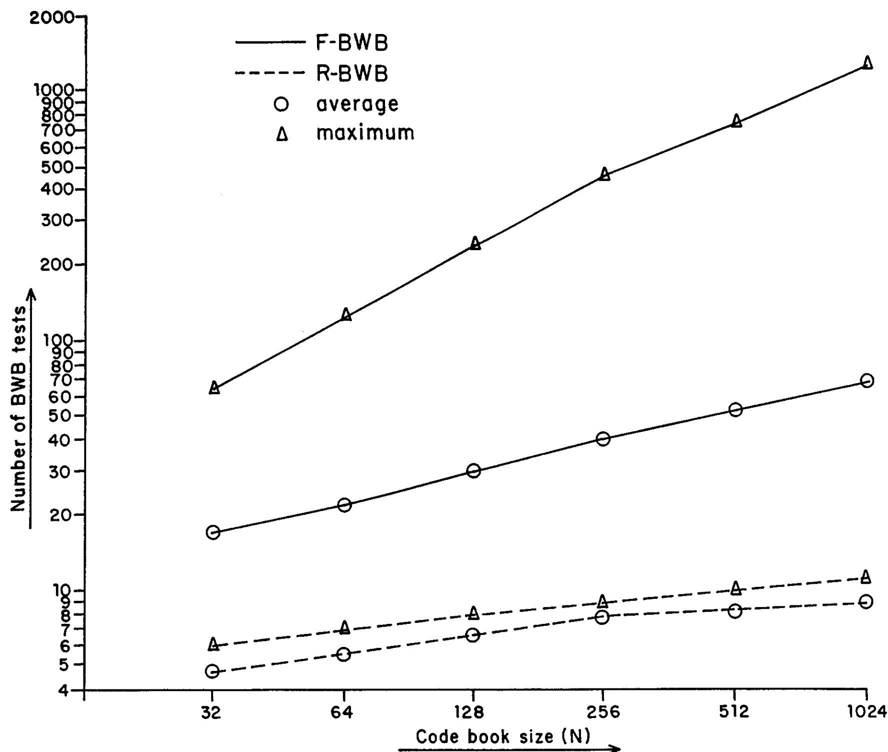


FIG. 12. Ball-within-bounds complexity for dimension $K = 8$; codebook sizes $N = 32$ to 1024; tree depth $d = \log N$.

R-BWB test is very low, with a maximum of only $d + 1$. Figure 12 compares the average and maximum F-BWB and R-BWB complexity for dimension $K = 8$ and codebook sizes $N = 32$ to 1024 with tree depth $\log N$. Figure 13 shows the average and maximum F-BWB and R-BWB complexity for codebook size $N = 1024$ and dimensions $K = 2$, 4, 6, 8, and 10 with tree depth 10. In both these cases, it can be seen that the F-BWB complexity is very high and R-BWB is significantly smaller with a very low average complexity and a worst-case complexity of $d + 1$. In comparison to the bounds-overlap-ball test, the complexity of the R-BWB test is negligible.

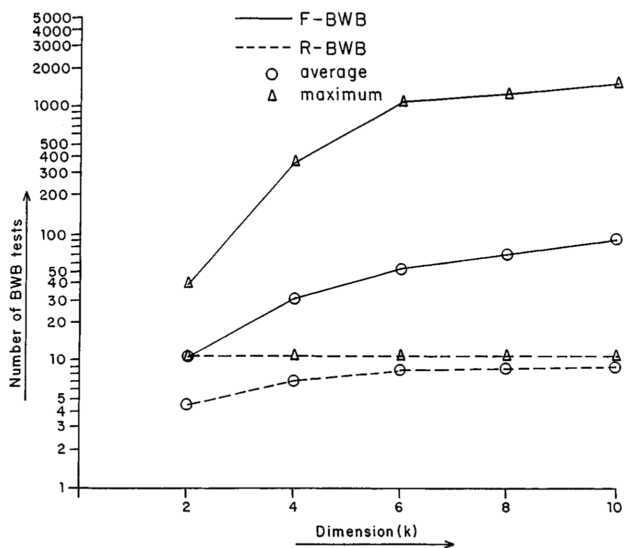


FIG. 13. Ball-within-bounds complexity dimensions $K = 2$, 4, 6, 8, and 10; codebook size $N = 1024$; tree depth = 10.

6. PRINCIPAL COMPONENT ROTATION

Here we show that the use of the principal component axes for data with a high degree of correlation across their components (such as speech waveform vectors) can enhance the optimization efficiency of the FBF optimization to yield reduced search complexity with the backtracking search. The FBF optimization chooses the optimal discriminant axis as the one along which the corresponding codevector components have the maximum variance. As mentioned earlier, the choice of axis by the maximum variance criterion at each node is based on the qualitative consideration of minimizing the probability of the current nearest-neighbor ball overlapping with the region on the other side of the partition. For a given current nearest-neighbor ball, this reduces the probability that the opposite bucket or the

sub-tree will have to be searched and indirectly reduces the average number of buckets which overlap the current nearest-neighbor ball.

As noted earlier, in vector quantization of speech waveform, the vectors have a high degree of correlation across their components. Therefore, the speech waveform vectors and the resultant codebook obtained from a typical training data do not have their maximum variance directions aligned along the coordinate axes. As a result, the projection variances of the codevectors on the coordinate axes are much smaller than along the main principal component directions. The variance of the codevector projections along the first principal component direction of the codevectors is maximum in comparison to any other direction. The variance is in general nonincreasing from principal coordinate direction 1 to K. The principal component rotation of the codevectors aligns the principal component directions along the coordinate axes. Subsequently, the variances of the codevector projections along the first few rotated coordinate axes will be larger than the variances on any of the unrotated coordinate axes. The principal component rotation will thus offer a more favorable set of coordinate axes for the FBF optimization in creating local partitions in which the probability that a nearest-neighbor ball will overlap with the opposite region will be less in comparison to the unrotated optimization. This will consequently reduce the average number of buckets overlapping the nearest-neighbor ball and the overhead complexity of backtracking search.

In this context of discussing the use of the principal component rotation, we note that in [11], Equitz employs the K-d tree with the FBF optimization and backtracking search for vector quantization of images. As a means of improving the efficiency of the K-d tree optimization and search, Equitz considers the use of "computed keys," where a reduced set of uncorrelated dimensions is obtained using the discrete cosine transform and the K-d tree is constructed and used for search in the reduced dimensional space. However, from simulation results, Equitz reports that the use of computed keys was found to degrade the performance of the encoding due to the approximation involved in using a reduced dimensional representation of the vectors and that there was only a very slight decrease in the computational complexity given the additional overhead of obtaining the "computed key" or the transformed vector. We note that the principal component rotation considered and studied here entails no approximation in the encoding since no dimensionality reduction is done, and the main objective is to provide a more favorable set of coordinate axes for improving the efficiency of the optimization and search. In this regard, we demonstrate that the principal component rotation significantly improves the efficiency of the FBF optimization and the associated backtracking search despite the additional cost of K multiplications per sample required for the principal component rotation of a test vector.

7. SIMULATION RESULTS WITH PRINCIPAL COMPONENT ROTATION

We now present some results showing the performance of the backtracking search with principal component rotation in the context of vector quantization encoding of speech waveform. First, we show a typical partitioning produced by the FBF optimization for speech waveform codebooks in 2 dimensions with principal component rotation. Figure 14 shows the partitioning for codebooks of size $N = 32$ and 1024 and vector dimension $K = 2$ with tree depth log N with principal component rotation. These can be compared with Fig. 6, which shows the partitioning of the same codebooks without rotation. (Henceforth, we refer to the principal component rotation as rotation and the regular coordinate axes without rotation as the unrotated axes.)

7.1. Optimization-Axis Histograms

We first show the effect of the principal component rotation in the way the optimal axis is chosen by the maximum-variance FBF criterion. Figure 15a shows the histograms of the axis used for partitioning by the FBF optimization for both the rotated and unrotated case. This was obtained for dimension $K = 8$ and codebook size $N = 1024$ by considering the partitioning axis at all the nodes of tree of depth 10. By comparing the histograms for the rotated and unrotated case, it can be observed that (i) all the K axes are chosen with more or less equal preference in the unrotated case indicating the uniformity of the component variances on the regular coordinate axes, and (ii) in the rotated case, the main principal component directions have a higher probability than any of the axes in the unrotated case and the FBF optimization has a strong preference for the main (lower) component directions in comparison to the higher ones. Due to the small variances of the higher principal component directions, the probability that these are chosen for optimization is lower than even any of the regular unrotated coordinate axes.

Curiously, it can be noted that the probability of the axis chosen is not maximum at the first principal

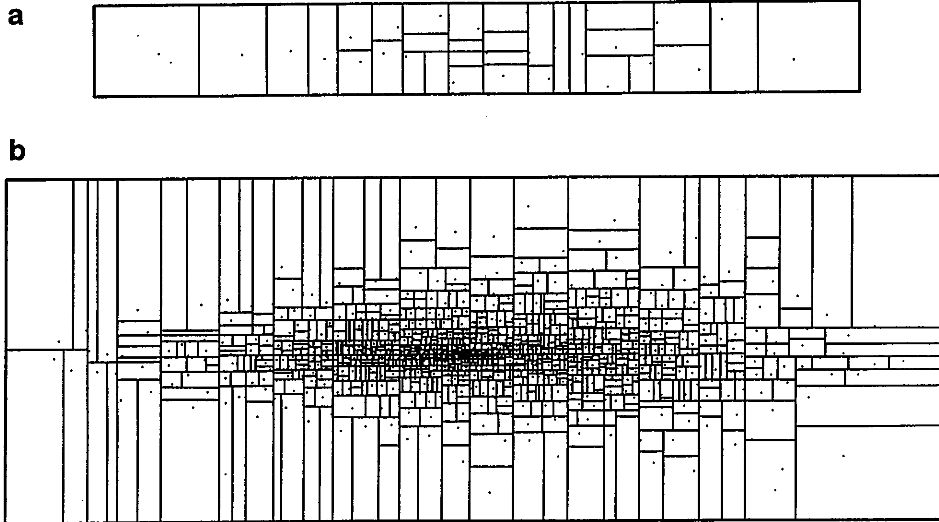


FIG. 14. K-d tree partitioning by FBF optimization with principal component rotation for dimension $K = 2$ and codebook sizes (a) $N = 32$ and (b) $N = 1024$.

component direction in the histogram shown in Fig. 15a. Instead, the maximum is located at the second principal coordinate axis. In general, we observe this to be typical for large dimensions—the axis used most frequently for optimization with rotation in the entire tree being one of the first few principal coordinate axes other than the first principal coordinate axis. An examination of the histogram of the axis chosen at each layer of the tree separately explains this general pattern of axis choice while revealing an interesting feature about the FBF optimization of the tree with rotation. Figure 15b shows, for the rotated and the unrotated case, the histograms of the axis chosen for the nodes at each layer starting from the root at depth $d = 1$ to the leaf (or bucket) layer at depth $d = 10$. It can be seen that, for the unrotated case, the variance of the codevector components, and hence the choice of axis, is more or less uniform over all the axes for all the layers. In comparison, for the rotated case, a specific pattern in the choice of the axis can be noted across the layers. Predictably, the first principal coordinate axis is chosen for partitioning the root. The partitioned regions are less likely to have the highest variance along the same axis and as a result, the partitioning in the second layer chooses the second principal coordinate axis which now has a relatively larger variance in each of the son regions of the root node. At the third layer, since the space has been partitioned along the second axis into regions with smaller variances, the optimization prefers the first axis for partitioning all the four regions at this layer. In general, the first few (lower) coordinate axes of high variance are favored with high probability in more or less a cyclic manner for increasing tree depths as their relative variances change with the partitioning at each layer. In the specific case shown, it can be observed that only the first three principal coordinate axes are used for optimization up to depth 6, and even for higher depths, the higher coordinates are used only very sparingly. Since the number of nodes increases with tree depth, and the lower principal coordinates other than the first coordinate axis are used with increasing probability for the nodes at larger tree depths, there is a relative abundance of one of the main principal coordinates over the first coordinate axis used in the optimization of the entire tree.

7.2. Search Complexity as a Function of Tree Depth

In Fig. 16, we compare the performance of the backtracking search for the rotated and unrotated case as a function of tree depth. Here, the results of the unrotated case is the same as shown in Fig. 7. For clarity of comparison, we show here only N_{cc}, N_{bob}, and the overall complexity $N_{cc} + N_{bob}$. It can be noted that both the average and the worst-case complexity of N_{cc} and N_{bob} show a reduction due to rotation at larger tree depths. In particular, the worst-case complexity shows a more significant reduction. This clearly indicates the advantage of the principal component rotation in reducing the main complexity due to the number of buckets overlapping with the nearest-neighbor ball and the overhead complexity of the backtracking search. In Fig. 17, we show the average and worst-case overall complexity

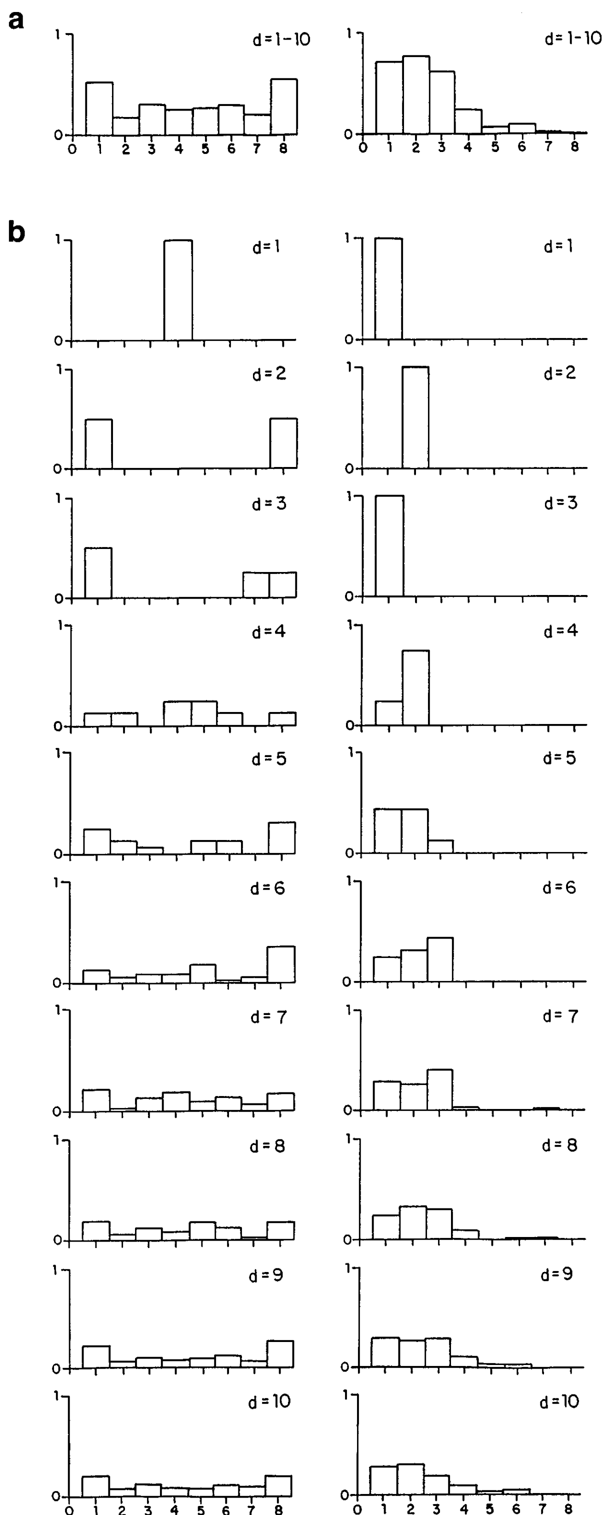


FIG. 15. Histograms of axis used for FBF partitioning without rotation and with principal component rotation, for dimension $K = 8$; codebook size $N = 1024$; tree depth 10; (a) at all nodes in tree depth 10; (b) at individual layers from depth 1 to 10.

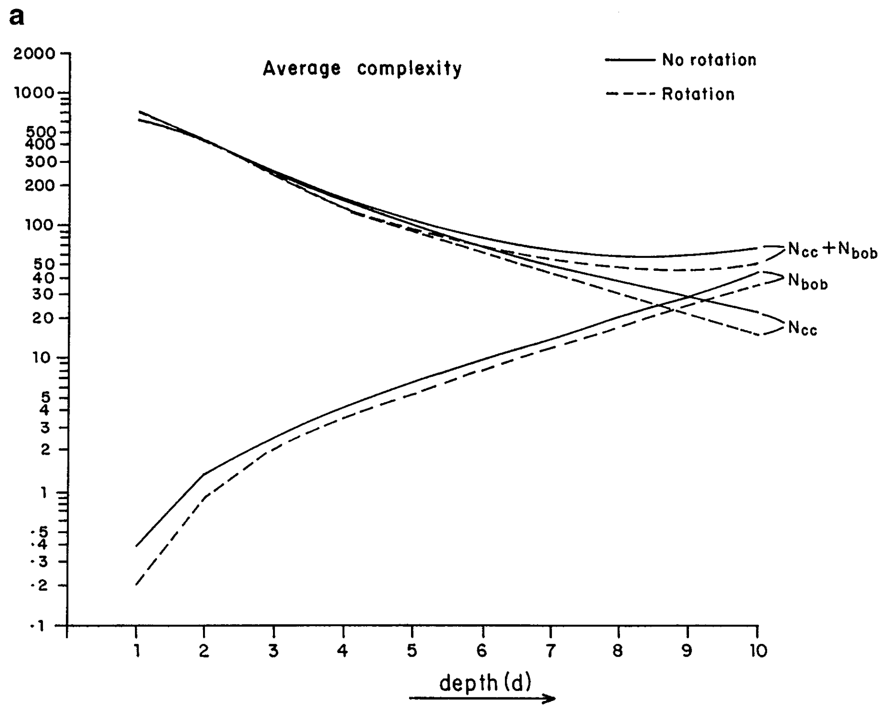
$N_{cc} + N'_{bob}$ for the rotated and the unrotated case as a function of tree depth. The significant reduction in the overall complexity due to rotation, particularly in the worst-case complexity, can be noted here.

7.3. Search Complexity for Various Codebook Sizes and Dimensions

In Fig. 18, we show the average and the worst-case overall complexity $N_{cc} + N'_{bob}$ for the rotated and the unrotated case for a fixed dimension $K = 8$ and codebook sizes $N = 32$ to 1024. In Fig. 19, we show the average and the worst-case overall complexity $N_{cc} + N'_{bob}$ for the rotated and the unrotated case for dimensions $K = 2$, 4, 6, 8, and 10 for codebook size $N = 1024$. In Figs. 18 and 19, it can be seen that while both the average complexity and worst-case complexity decrease due to rotation, the reduction in the worst-case complexity is very significant. Particularly, these results show a trend of increased reduction in the average and worst-case complexities due to rotation for increasing codebook size and dimension. The principal component rotation incurs an overhead cost of K^2 multiplications per vector, equivalent to K distance computations per vector. However, the complexity reduction due to the rotation is seen to more than compensate this overhead cost of rotation. The principal component rotation is therefore clearly a very useful technique for improving the efficiency of the FBF optimization and in reducing the complexity of the backtracking search. In particular, the results shown clearly demonstrate this for vector quantization of speech waveform for large dimensions and codebook sizes.

7.4. Probability of No Backtracking

We now show a measure which reflects the effect of principal component rotation in reducing the backtracking search overhead. Figure 20 shows the probability that the test vector is *encoded at the leaf layer without backtracking,* i.e., the probability that the search terminates within the bucket containing the test vector after examining the codevectors inside that bucket at depth d for a tree of depth d. This represents the most efficient search situation, as the search involves no backtracking. This probability is shown for dimension $K = 8$ and codebook size $N = 1024$, for tree depth 1 to 10. It can be seen that, in general, the probability of the search terminating without any backtracking is very high for tree depth 1 and decreases rapidly as the depth of the tree increases, clearly reflecting the basic backtracking behavior of the search as the bucket size becomes smaller. For large bucket sizes, the search finds the nearest neighbor within the bucket containing the



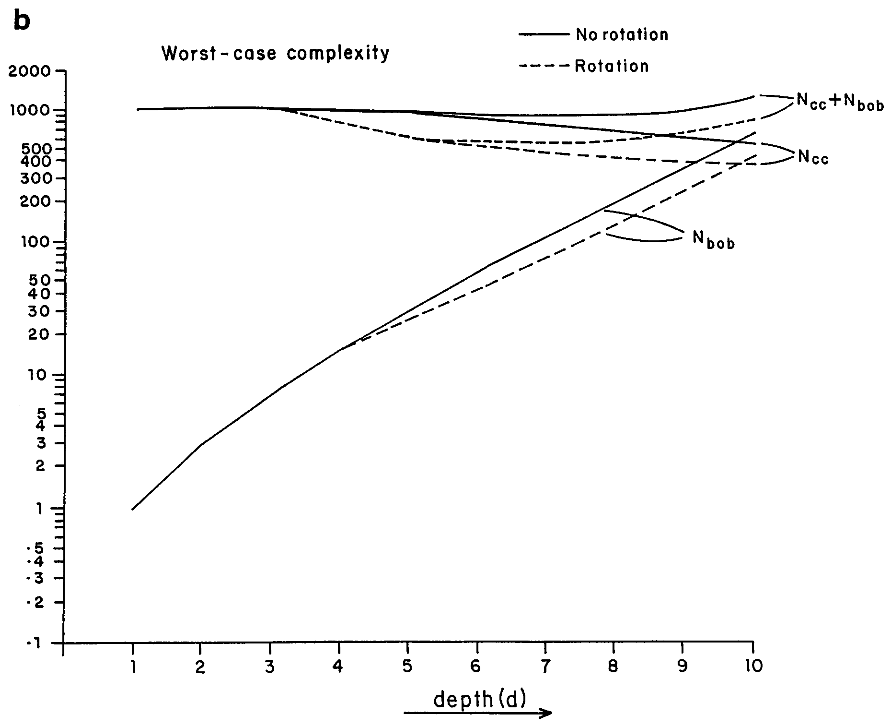


FIG. 16. (a) Average and (b) worst-case complexities of backtracking search with and without rotation for tree-depth 1 to 10 for codebook size $N = 1024$ and dimension $K = 8$; No Rotation: without principal component rotation; Rotation: with principal component rotation.

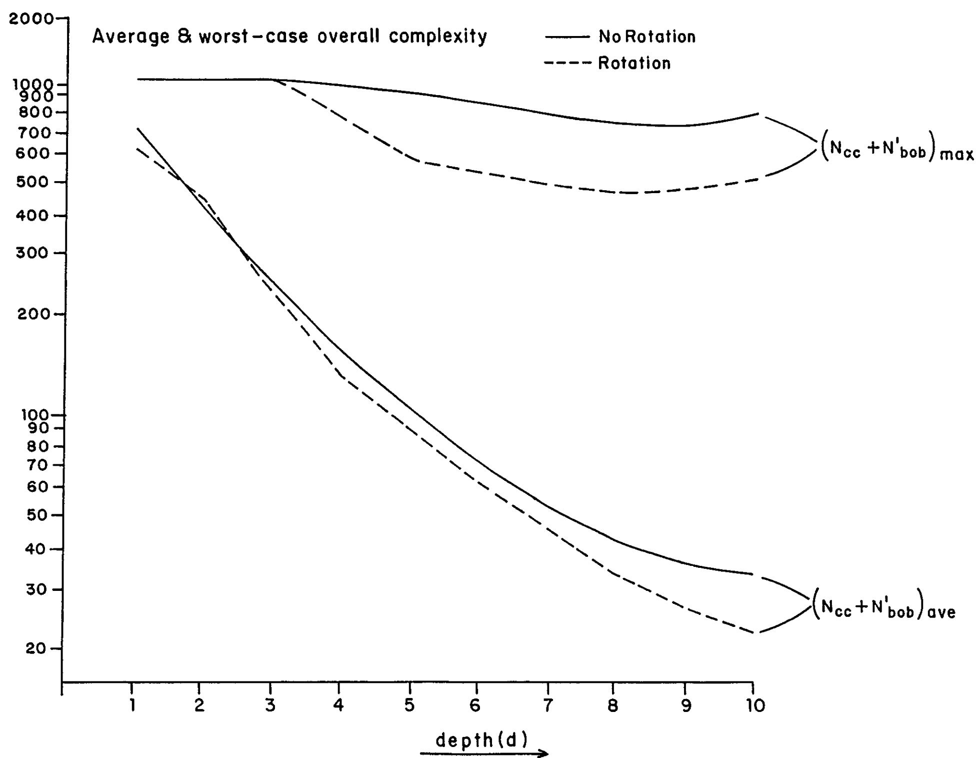


FIG. 17. Average and worst-case overall complexities of backtracking search with and without rotation for tree-depth 1 to 10 for codebook size $N = 1024$ and dimension $K = 8$; No Rotation: without principal component rotation; Rotation: with principal component rotation.

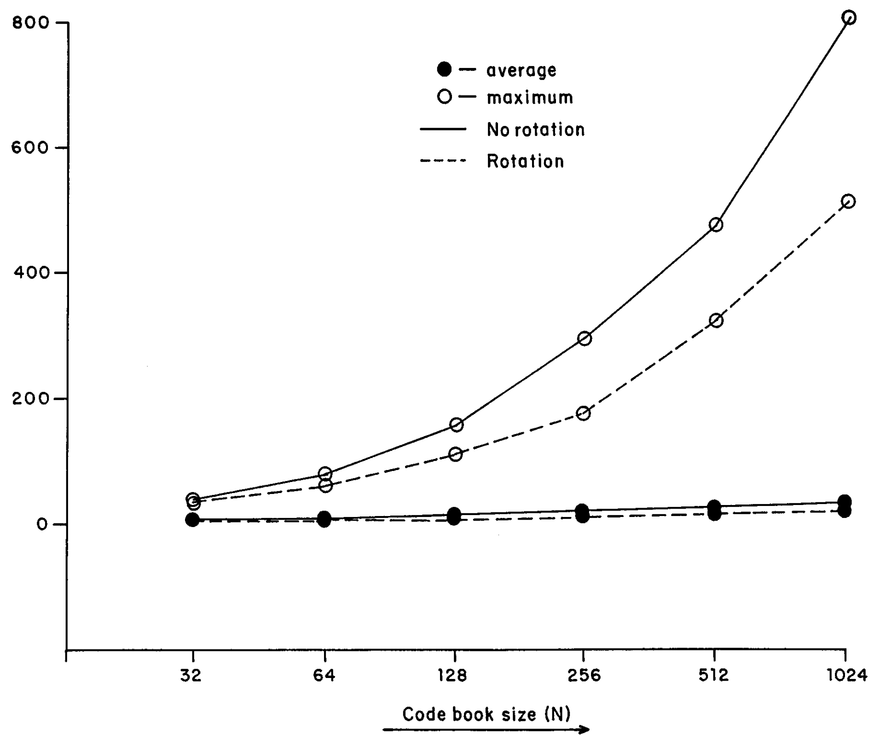


FIG. 18. Average and worst-case overall backtracking search complexities with and without rotation for codebook sizes $N = 32$ to 1024; dimension $K = 8$; tree depth $d = \log N$; No Rotation: without principal component rotation; Rotation: with principal component rotation.

test vector and the ball-within-bounds is also satisfied with a very high probability within this bucket. This decreases with the decrease in the bucket size forcing the search to backtrack to examine adjacent buckets. However, it can be observed from Fig. 20 that the principal component rotation increases this probability substantially relative to the probabilities for the unrotated case; for instance, the probability increases from about 0.6 to 0.8 for a tree of depth 1 and from 0.003 to 0.016, by nearly a factor of 5, for a tree of depth 10.

8. DISCUSSION AND CONCLUSIONS

In this paper the optimization and backtracking search algorithm proposed by Friedman *et al.* [15] have been considered in detail. The basic algorithm is described; specifically the features of optimization as observed from theoretical analysis and from the empirical performance of the backtracking search are highlighted. The performance of the FBF optimization and backtracking search is studied for various dimensions, codebook sizes, and tree depths in the context of vector quantization encoding of speech waveform. The high search overheads and poor worst-case efficiency of the backtracking search have been demonstrated using statistics of the backtracking search overheads such as the bounds-overlap-ball test, ball-within-bounds test, and backtracking depth. It is seen that despite the 2^K dependence of the theoretical average complexity bound, the actual average complexity of the backtracking search in vector quantization encoding of speech waveform is excellent. However, it is shown that the backtracking search has a very high worst-case computational overhead and that this may be unacceptable in many practical applications such as real-time vector quantization encoding, where the worst case complexity is of considerable importance in addition to average computational complexity.

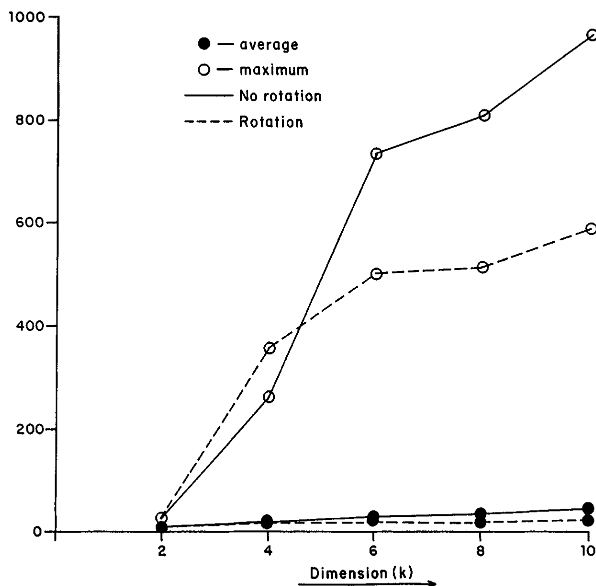


FIG. 19. Average and worst-case overall backtracking search complexities with and without rotation for dimensions $K = 2$, 4, 6, 8, and 10; codebook size $N = 1024$; tree depth = 10; No Rotation: without principal component rotation; Rotation: with principal component rotation.

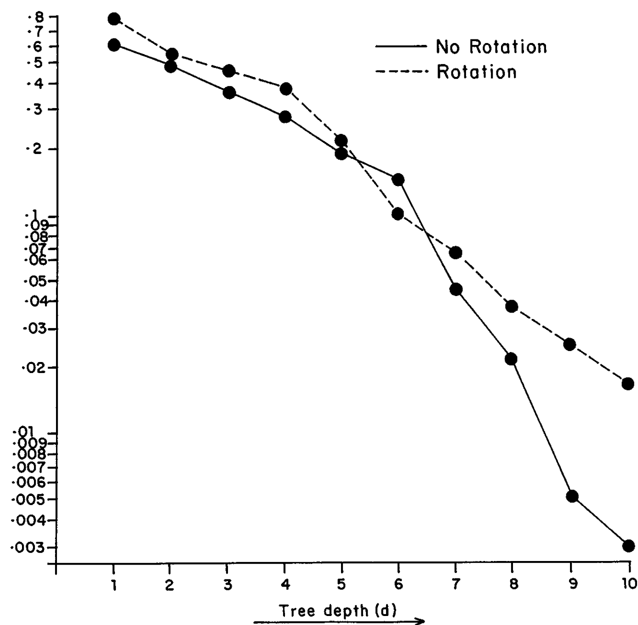


FIG. 20. Probability of search terminating at leaf layer (no backtracking) with and without rotation for tree depth $d = 1$ to 10; dimension $K = 8$ and codebook size $N = 1024$; No Rotation: without principal component rotation; Rotation: with principal component rotation.

It is also shown that the ball-within-bounds test needs to be done only at the ancestor nodes of the bucket containing the test vector, rather than at every nonterminal node whose sub-tree has been completely examined and that this reduces the worst-case complexity of the ball-within-bounds test from $O(2^d)$ to $O(d)$ for tree depth d. We have studied the use of principal component rotation in the context of vector quantization of speech waveform where there is a high degree of correlation across the components of a vector and shown that this can improve the efficiency of optimization and reduce both the main search complexity and the overhead complexity of the backtracking search.

One of the applications where Friedman *et al.*'s [15] optimization can be appropriately applied is in

multidimensional nonparametric density estimation and classification problems. In this case, under conditions when the size of the data set $N \gg 2^K$ and N can be independent of K, the average complexity performance of the backtracking search algorithm (despite its $O(2^K)$ dependence of average search complexity) can be considered satisfactory over the conventional full-search complexity.

To provide a qualitative insight on the underlying principle of bucketing techniques and complexity of the K-d tree based backtracking search, we note the following extract [2] on search based on bucketing techniques:

> In the most general terms, 'buckets' mean the subregions into which the entire region under consideration is partitioned. . . . The basic properties of buckets that are exploited in reducing the complexity of the search are:
>
> 1. If buckets have a simple shape, i.e., if they are square or rectangular with the sides parallel to the coordinate axes, it is computationally quite easy to partition the region under consideration into the buckets and to determine to which bucket a point belongs. . . .
>
> 2. Unless the distribution of geometrical objects is extremely skewed, the number of objects belonging to a bucket is usually bounded by a constant if the entire region is partitioned into a number of buckets equal in order to the number of objects.
>
> 3. In most geometrical problems, the mutual influence between two objects is likely to decrease with their distance, so that it is usually a good strategy to first construct a local or myopic solution and then to combine local solutions to make a global solution.
>
> If we can take advantage of all these properties of buckets, we shall be led to an algorithm which runs in linear time on the average. However, a simple-minded use of buckets will not give a satisfactory result in many problems. It is crucial for the success of an algorithm using bucketing techniques whether or not we can devise a well-tuned algorithm for processing the information within each bucket as well as that for combining the local results into a global solution. This last part is highly problem dependent.

The backtracking search follows this approach of finding a local solution and carrying out a search in the other buckets to find the global solution, moving from one bucket to the other with the help of the tree. This search is based on the idea of finding a codevector nearer to the test vector than the current nearest neighbor. As a result, the nearest-neighbor ball provides the search constraints and becomes the primary geometric construct controlling the search. The K-d tree structure essentially helps in obtaining a reasonably small initial nearest-neighbor ball and in guiding the search to neighboring buckets; in this process, the verification of the optimality of the solution dominates the rest of the search and this becomes the primary reason for the worst-case inefficiency of the backtracking search.

In this context, we note that an alternate search paradigm which exploits the space localization property of the K-d tree structure more efficiently is to characterize the Voronoi partition of the search space in terms of "bucket–Voronoi intersections." Central to the idea of the bucket–Voronoi intersection search framework is the notion of the Voronoi partition of the given set of codevectors. Given the Voronoi partition, the problem of nearest-neighbor search essentially reduces to the point-location problem of determining which Voronoi region contains the test vector; i.e., if the test vector $\mathbf{x}$ is contained in a Voronoi region $\mathbf{V}_j$, the associated codevector $\mathbf{c}_j$ will be the nearest neighbor of $\mathbf{x}$. In the bucket–Voronoi intersection search framework, each leaf (or bucket) of the K-d tree is associated with a set of codevectors whose Voronoi regions intersect with the bucket region. The search involves first locating the bucket containing the test vector and subsequently performing a full search only among the small set of codevectors associated with the bucket without any further backtracking.

As viewed in the framework of the bucketing principle, this represents a shift in the primary geometric construct which localizes the search space from the current-nearest-neighbor ball (centered at the test vector) to the Voronoi regions (defined with respect to the codevectors) intersecting with the bucket region containing the test vector. The bucket–Voronoi intersection search framework provides an efficient bucketing technique in which the explicit consideration of the Voronoi information localizes the search within one bucket and results in an efficient alternative to the backtracking search where the neighboring buckets have to be examined for obtaining the correct global solution for nearest-neighbor search. Thus, the bucket–Voronoi intersection based search procedure is simple and direct and can offer significant complexity reduction over the backtracking search which has high computational overheads and worst-case complexity.

The above extract [2] also points to the importance of efficient optimization of the bucket partition for a given set of objects and search framework. In addition, the fact that the number of objects belonging to the bucket can usually be bounded by a constant (if the search space is partitioned into a number of buckets equal in order to the number of objects) provides a direct motivation for achieving constant search complexity with the bucket–Voronoi intersection search, where the objects within a bucket correspond to the Voronoi region intersections with the bucket region and the search complexity is related only to the number of Voronoi regions intersecting with the bucket. This permits optimization of the K-d

tree in terms of bucket–Voronoi intersections which relate more directly to the search complexity than in the case of the backtracking search. The design of the *K*-d tree under the bucket–Voronoi intersection framework thus considers the Voronoi information explicitly and this issue has been addressed in detail in [8, 21, 26–29], where different optimization criterion and procedures of the *K*-d tree have been proposed and studied to minimize the complexity of the search based on bucket–Voronoi intersections.

REFERENCES

1. Anderson, D. P. Techniques for reducing pen plotting time. *ACM Trans. Graphics* **2**(3) (1983), 197–212.
2. Asano, T., Edahiro, M., Imai, H., and Iri, M. Practical use of bucketing techniques. In *Computational Geometry,* (G. T. Toussaint, Ed.). Elsevier, Amsterdam,1985, pp. 153–195.
3. Baird, H. S. Applications of multi-dimensional search to structural feature identification. In *Syntactic and Structural Pattern Recognition* (G. Ferrate *et al.,* Eds.), NATO ASI Series, Vol. F45, Springer-Verlag, Berlin/New York, 1988, pp. 137–149.
4. Bentley, J. L. Multidimensional binary search trees used for associative searching. *Comm. ACM* **18**(9) (1975), 509–517.
5. Bentley, J. L., and Friedman, J. H. Data structures for range searching. *Comput. Surveys* **11** (1979), 397–409.
6. Bentley, J. L., Weide, B. W., and Yao, A. C. Optimal expected-time algorithms for closest point algorithms. *ACM Trans. Math. Software* **6**(4) (1980), 563–580.
7. Buzo, A., Gray, A. H., Jr., Gray, R. M., and Markel, J. D. Speech coding based upon vector quantization. *IEEE Trans. Acoust. Speech Signal Process.* **ASSP-28**(5) (1980), 562–574.
8. Cheng, D. Y., and Gersho, A. A fast codebook search algorithm for nearest-neighbour pattern matching. In *Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing,* 1986, pp. 6.14.1–6.14.4.
9. Eastman, C. M. Optimal bucket size for nearest-neighbour searching in *k-d* trees. *Inform. Process. Lett.* **12**(4) (1981), 165–167.
10. Eastman, C. M., and Zemankova, M. Partially specified nearest neighbor searches using *k-d* trees. *Inform. Process. Lett.* **15**(2) (1982), 53–56.
11. Equitz, W. H. *Fast algorithms for vector quantization picture coding.* Master's thesis, MIT, June 1984.
12. Equitz, W. H. Fast algorithms for vector quantization picture coding. In *Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing,* 1987, pp. 725–728.
13. Equitz, W. H. A new vector quantization clustering algorithm. *IEEE Trans. Acoust. Speech Signal Process.* **37**(10) (1989), 1568–1575.
14. Silva Filho, Y. V. Optimal choice of discriminators in a balanced *K-d* binary search tree. *Inform. Process. Lett.* **13**(2) (1981), 67–70.
15. Friedman, J. H., Bentley, J. L., and Finkel, R. A. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Software* **3**(3) (1977), 209–226.
16. Gersho, A., and Cuperman, V. Vector quantization: A pattern matching technique for speech coding. *IEEE Commun. Mag.* **21** (1983), 15–21.
17. Gray, R. M. Vector quantization. *IEEE ASSP Mag.* **1** (1984), 4–29.
18. Juang, B. H., and Gray, A. H., Jr. Multiple stage vector quantization for speech coding. In *Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing,* 1982, pp. 597–600.
19. Knuth, D. E. *Sorting and searching: The art of computer programming III.* Addison–Wesley, Reading, MA, 1973.
20. Linde, Y., Buzo, A., and Gray, R. M. An algorithm for vector quantization design. *IEEE Trans. Commun.* **COM-28** (1980), 84–95.
21. Lowry, A., Hossain, Sqama, and Millar, W. Binary search trees for vector quantization. In *Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing,* 1987, pp. 51.8.1–51.8.4.
22. Makhoul, J., Roucos, S., and Gish, H. Vector quantization in speech coding. *Proc. IEEE* **73** (1985), 1555–1588.
23. Moayeri, N. *Fast Vector Quantization.* Ph.D. thesis, University of Michigan, 1986.
24. Moore, A. W. *Efficient Memory-Based Learning for Robot Control.* Tech. Rep. 209, University of Cambridge Computer Laboratory, University of Cambridge, Nov. 1990.
25. Paliwal, K. K., and Ramasubramanian, V. Vector quantization in speech coding: A review. *Indian J. Technol.* **24** (1986), 613–621.
26. Ramasubramanian, V. *Fast Algorithms for Nearest-Neighbour Search and Application to Vector Quantization Encoding.* Ph.D. thesis, Computer Systems and Communications, School of Physics, TIFR, Bombay, India, 1991.
27. Ramasubramanian, V., and Paliwal, K. K. An optimized *K-d* tree algorithm for fast vector quantization of speech. In *Signal Processing IV: Theories and Applications, Proc. EUSIPCO '88 Grenoble, France,* (J. L. Lacoume *et al.,* Eds.), North-Holland, Amsterdam/New York, 1988, pp. 875–878.
28. Ramasubramanian, V., and Paliwal, K. K. A generalized optimization of the k-d tree for fast nearest-neighbour search. In *Proceedings of 4th IEEE Region 10 International Conference, TENCON '89,* 1989, pp. 565–568.
29. Ramasubramanian, V., and Paliwal, K. K. Fast *K*-d tree algorithms for nearest-neighbour search with application to vector quantization encoding. *IEEE Trans. Acoust. Speech Signal Process.* **40**(3) (1992), 518–531.
30. Sabin, M. J., and Gray, R. M. Product code vector quantizers for speech waveform coding. In *Conference Record Globecom '82,* 1982, pp. 1087–1091.
31. Sabin, M. J., and Gray, R. M. Product code vector quantizers for waveform and voice coding. *IEEE Trans. Acoust. Speech Signal Process.* **ASSP-32**(3) (1984), 474–488.
32. Wan, S. J., Wong, S. K. M., and Prusinkiewicz, P. An algorithm for multidimensional data clustering. *ACM Trans. Math. Software* **14**(2) (1988), 153–162.