Complexity Analysis for Partitioning Nearest Neighbor Searching Algorithms

Pierre Zakarauskas and John M. Ozard

Abstract—In this paper we present cost estimates for finding the k-nearest neighbors to a test pattern according to a Minkowski p-metric, as a function of the size of the buckets in partitioning searching algorithms. The asymptotic expected number of operations to find the nearest neighbor is presented as a function of the average number of patterns per bucket n and is shown to contain a global minimum.

Index Terms—k-nearest-neighbor search, nearest-neighbor search, complexity analysis, cost analysis, Minkowski p-metric, k-d tree partitioning, ordered partitioning, product partitioning.

1 INTRODUCTION

The problem of finding which among a set of stored patterns are closest to a given test pattern is of great general interest. The partial justification for this was obtained by Fix and Hodges [1], [2] who showed that classification on the basis of the class most heavily represented among the k-nearest neighbors (kNN) to the test pattern is asymptotically Bayes’ risk efficient. Cover and Hart [3] showed that the nearest neighbor (NN) rule leads to a misclassification risk which is between ρ and 2ρ, where ρ is the Bayes’ risk. The Bayes’ risk is the minimum cost of a decision based on the Bayes’ criterion. The lower limit is reached when ρ → 1, i.e., for very noisy data. The NN search, being a nonparametric method, does not depend in any way on knowing the underlying probability distribution function (pdf) of the patterns. Because the direct comparison of every stored pattern with the test pattern is cost prohibitive for a large number of stored patterns N, considerable research has been invested in finding efficient search methods.

Several Nearest Neighbors Search Algorithms (NNSAs) have been published. It was recognized in the 1970’s that restricting the search to the neighborhood of the test pattern leads to a decrease in search time. Two main classes of algorithms exist based on whether or not the search space is a vector-space, i.e., whether the data points can be represented by a constant set of numbers. If the search space is not a vector-space, but the dissimilarity measure is a metric, then NNSAs based on the triangle inequality are still applicable [4], [5], [6], [7]. In the following, only partitioning algorithms will be considered (for a review of the field, see [8]).

Partitioning algorithms pre-process the data by partitioning the search space and distributing the patterns in the resulting “buckets” according to the values of their components, sometimes referred to as “keys.” There are two major types of partitioning used: hierarchical and product partitioning. In product partitioning, introduced by Shen and Lee [9], each partition subdivides the search space for the range of values along that dimension. The resulting partitioning thus defines a grid. Product partitioning is best suited for the case where the data are distributed independent in each dimension. In hierarchical partitioning on the other hand, partitions only divide the search space locally.

It is not sufficient to look up only the one bucket whose keys fit that of the test pattern, as is done for a discrete search (hash table). For a uniform rectangular distribution of stored patterns, the probability of the NN of being in the same bucket as the test pattern is estimated in [10] to be

\[ p_{NN} = \left(1 + \frac{n}{N}ight)^{-d}, \]

where n is local average number of patterns per bucket. The estimate is obtained by assuming a regularly-spaced distribution of patterns. \( p_{NN} \) falls rapidly with the dimensionality of the search space. For example, with 100 patterns per bucket, in a three-dimensional space, \( p_{NN} \) is 0.56, while it falls to only 0.33 in a four-dimensional space, and to 0.19 for a five-dimensional space. Therefore, one must perform an expanding search centered on the test pattern, and look into several neighboring buckets, in order to guarantee that the NN is found. Friedman et al. [11], Shen and Lee [9], and Kim and Park [12] describe algorithms based on hierarchical partitioning of the search space which finds the NN in a time roughly independent of N. If the size of the buckets is adjusted so that each contains about the same number n of patterns, then the performance of these algorithms is also independent of the pattern distribution.

The distance between two points can be defined in terms of one of a variety of metrics. The Minkowski p-metric is a measure of the distance \( D_p \) between two points, and has been used extensively in experimental sciences. It is defined as

\[ D_p(x, y) = \left( \sum_{i=1}^{d} |x_i - y_i|^p \right)^{1/p}. \]

The surface defined by all points equidistant from a center point is called a p-sphere. The p = 1 metric, or \( L_1 \) metric, is often called the “city block distance,” and its p-sphere is a d-dimensional diamond. The \( L_2 \) metric is the Euclidean distance, and its p-sphere is a d-dimensional hypersphere. The other often-used metric is the \( L_\infty \) metric, sometimes called the “max metric,” and the corresponding p-sphere is a hypercube.

When choosing or implementing an algorithm, it is very useful to know its cost in terms of storage and computation. This allows the optimization of the implementation by selection of the best values of the algorithm’s free parameters. Such a complexity analysis has been provided for some NNSAs. Yunk [13] derives a complexity analysis for an algorithm searching with each key separately for the kNN using the \( L_\infty \) norm (the cube algorithm).

Friedman et al. [11] provide a complexity analysis of the k-d tree NN search, and derive an explicit calculation for the number of buckets explored, again using the \( L_\infty \) norm as a distance measure.

Eastman [14] derives the number of buckets from the Friedman et al. calculation for the \( L_\infty \) norm, and gives a general form for a bound on the optimal bucket size for k-d tree search for any p. However, the result contains a proportionality constant which is left unspecified. Finally, Kim and Park [12] derive an upper bound for the number of buckets searched for the \( L_2 \) (Euclidean) norm. In this case they were examining a faster hierarchical-search algorithm than that of Friedman et al. which they call “ordered partition search.” A good review of the subject of fast search of partitioned files is provided by Eastman [15].

In the present paper, we derive a complexity analysis when the distance measure used is any Minkowski p-metric, for either product or hierarchical partitioning algorithms, and we provide a direct way to find the optimum number of patterns per bucket. The usefulness of the new results lies with the popularity of the finite-p Minkowski metric, in particular the Euclidean metric. Since the results depend on p, the results obtained for the \( L_\infty \) norm...
cannot be used for the finite-$p$ norms. The results contained in this paper allow potential users of NNSAs to estimate the final performance of the algorithms with their particular problem and resources without having to run large numbers of tests to do so. It also allows the user to compare the performance of different NNSAs before setting choice on one without having to implement them all first.

Section 2 reviews three popular partitioning algorithms for which we will derive explicit performance estimates in Section 3. Section 4 reports the results of a computer simulation, while Section 5 contains a discussion of the results and the conclusions.

2 SEARCH ALGORITHMS

In brief, NNSAs rely on the following idea. The search space is first partitioned, and one sorts the stored patterns into buckets. When a test pattern is supplied, one compares it to all patterns within the bucket containing the test pattern. If the distance $D_{mn}$ between the test pattern and the $k$th closest pattern is less than the distance to the closest boundary, then the search stops. If not, one performs the search within an expanding $p$-sphere centered on the test pattern, until $k$ patterns are found that fall within that $p$-sphere. Let the set of stored patterns be $\{a_i | i = 1, \ldots, N\}$, with each $a_i$ being a $d$-dimensional vector in $\mathbb{R}^d$. Given a test pattern $a_T$, the problem is to find the $k$ nearest patterns $a_1, a_2, \ldots, a_k$ in the shortest time possible. In some cases, the patterns $a_i$ may be stored in a mass storage medium, and fetching them involves a greater cost than making the distance calculation itself. The total cost is to be minimized, and the optimum solution thus depends on the implementation.

The preprocessing stage consists of finding which patterns (if any) are contained in each bucket. This information is stored as a list of pattern numbers associated with each bucket. The preprocessing can be performed in $O(N)$, since determining which bucket a pattern falls in does not depend on the other patterns' location. We will now discuss in more detail three types of partitioning algorithms.

2.1 $k$-$d$ Tree Partitioning

A $k$-$d$ tree is a binary tree that divides the subspace at each node with a boundary placed for one of the keys. In [11], Friedman et al. argue that the maximum amount of information is gained by placing each new partition in the dimension with the largest spread in values, and place it at the median of the coordinates of the patterns along that dimension. The search proceeds up the tree from the bucket containing the test pattern. Nodes on a given level are explored if their distance to the test pattern is less than $D_{max}$.

2.2 Ordered Partitioning

Kim and Park [12] introduced the ordered partitioning algorithm, and their simulations show that it is capable of even better performance than that of the $k$-$d$ tree partitioning algorithm. The ordered tree has depth $d - 1$, one level for each of the first $d - 1$ keys. Each level contains a predetermined number $n_i$ of partitions for each key. The partitions are placed in increasing order of the key at each node. As in the $k$-$d$ tree, the terminal nodes are the patterns themselves. Buckets are defined by the values of the $d - 1$ partitions in the nodes above each pattern. The difference in the value of the last key is checked, and if it is less than $D_{max}$, then the distance is calculated. This limits the search to a hypercylinder whose axis is centered on the search $p$-sphere.

2.3 Product Partitioning

Product partitioning is most appropriate for the case where the pdf of the patterns, $g(x)$, is separable, i.e., the patterns are uncorrelated. This means that the function $g(x)$ can be expressed as the product of the distribution functions along each of the coordinates:

$$g(x) = g_1(x_1) g_2(x_2) \cdots g_d(x_d).$$

The assumption of uncorrelated data is not as restrictive as it may seem, because correlated data can often be reduced to uncorrelated data through a rotation of the axis, a method known as principal component analysis [16].

Optimal product partitioning is accomplished by specializing the procedure for the $k$-$d$ tree given in Section 2.1 to the case of an uncorrelated density distribution function. For product partitioning, the optimal amount of information is gained by placing partitions along each dimension so that the same number of patterns are contained between partitions, and therefore the average number $n$ of patterns per bucket is constant. Arbitrarily selecting the origin as a starting point, partitions are placed for every integer value of the cumulative distribution function $G_p$ normalized by $\frac{n}{d}$:

$$G_p = \frac{n}{d} \int_0^x g(x)dx.$$

The ensemble of partitions defines a grid onto the search space.

One would expect that it would be easier to select the buckets that are to be searched with product partitioning than with hierarchical partitioning, since each partition extends across the whole domain. Along each dimension, the range of buckets to be searched is found by selecting the slice of the $p$-sphere whose distance to the test pattern is less than or equal to $r$. This range calculation needs to be done once only for each slice. An algorithm different from that presented in [9] for search with product partitioning of the search space is presented in the Appendix. This algorithm will be used in the simulations of Section 4. The search sphere is expanded by a factor $\tau_{sec}$ until a pattern is found within the sphere. Note in particular the recursive call to the function SearchSphere($j, r$), which determines which buckets to search within an hypersphere slice of dimension $j$ having radius $r$. Along each slice, the indices of the first and last buckets to be searched are calculated, and then this slice is in turn sliced along the $(j - 1)$th dimension, until $j = 1$, where the slices constitute columns of buckets. The main difference between this algorithm and the one presented in [9] is the form of backtracking.

3 ANALYSIS OF PERFORMANCE

What is the optimum bucket size? Or, equivalently, what is the optimum number $n$ of patterns per bucket? For a given dimension $d$ of the search space, $n$ is the only free parameter of the algorithms. There must exist an optimal value of $n$ since, if $n$ grows very large, many more patterns must be matched than only those contained in the search $p$-sphere. On the other hand, if $n$ is very small, the number of buckets becomes very large and the overhead of selecting the buckets increases. We present below an expression for the expected number of calculations to find the kNN, and find the optimum bucket size.

3.1 Expected Radius of $p$-Sphere

After the initial bucket has been searched, a minimum $p$-sphere radius should be chosen so that the probability $q_i$ of finding $i$ patterns within that $p$-sphere is appreciable, but not too great, say between 0.1 and 0.9. The hypervolume of the search $p$-sphere is

$$V_r = K_{ps} r^d,$$

where $K_{ps}$ is a constant determined by $p$ and $d$. For the three most used metrics ($p = 1, 2, \infty$), this constant is

$$K_{ps} = \begin{cases} 2^d & p = 1, 2, \infty \end{cases}.$$
\[ K_{z;d} = \frac{\pi^{d/2}}{\Gamma(d/2 + 1)}. \]

(a derivation for \( K_{z;d} \) is provided in [17]), and
\[ K_{0;0} = 2^d. \]

Once a search radius \( r \) has been selected to give \( k \) patterns with a certain probability, one must retrieve the patterns within all the buckets which partition the search \( p \)-sphere centered on the target pattern, and test for the distance of all of them to the test pattern. If the distance \( D_{\text{min}} \) of the \( k \)th nearest pattern found at that point is not smaller than the radius \( r \) of the search \( p \)-sphere, the hypervolume of the search \( p \)-sphere is increased by a small factor. \( D_{\text{min}} \) is updated after searching every bucket, and the search stops when \( D_{\text{min}} \leq r \).

The initial value of \( r \) should be close to the expected value of the distance between a random pattern and its \( k \)NN. This is found by taking the expected value of the radius of the hypersphere containing exactly \( k \) \(- 1 \) patterns surrounded by a shell of thickness \( \Delta r \) containing at least one pattern, assuming a Poisson distribution of the number of patterns within a given hypervolume [10].

\[ E[r] = \left( K_{r;d} \right)^{-1/d} \left( k - 1 \right)! \Gamma \left( \frac{d + 1}{d} \right). \]

### 3.2 Number of Buckets

The average number of buckets touched by a \( p \)-sphere of radius \( r \) is the product of the probability of the \( p \)-sphere touching any particular bucket and the total number of buckets. Let us start by calculating the probability \( \Omega_i \) for a \( p \)-sphere of radius \( r \) touching a given bucket in an unbounded or a wrap-around \( d \)-dimensional space. In a particular neighborhood, one can consider the size of the buckets as locally constant [11]. The length of a bucket along dimension \( i \) will be denoted by \( w_i \). Without loss of generality, we take the total hypervolume of the search space to be unity, thus making the probability \( \Omega_i \) equal to the hypervolume contained within a distance \( r \) to any side of the bucket. In two-dimensional space, this probability is equal to the area
\[ \Omega_i = w_i w_{i+1} + 2r(w_i + w_{i+1}) + \pi r^2. \]

The local density of buckets is \( (w_i w_{i+1})^{-1} \), so that the number of buckets partitioning the \( p \)-sphere is
\[ B_i = \frac{\Omega_i}{w_i w_{i+1}} = 1 + \frac{2r(w_i + w_{i+1})}{w_i w_{i+1}} + \pi r^2. \]

One useful conclusion one can draw from this expression for \( B_i \) is that, for a given bucket volume \( \sigma_i \), the minimum of \( B_i \) occurs for \( w_i = w_{i+1} \), i.e., for square buckets. This conclusion holds generally in \( d \)-dimensions. In \( d \)-dimensional space, the average number \( B_d \) of buckets partitioning a \( p \)-sphere of radius \( r \) is found by generalizing the derivation of \( B_i \) to \( d \) dimensions [10]:
\[ B_d = \frac{d}{i=0} K_{r;0} r^d H_{i,d} \]

where we have used the shorthand
\[ H_{i,d} = \left[ \frac{1}{i!} \sum_{k=1}^{d} \frac{(i-\delta_{i1})^{k-2}}{k!} \prod_{j=1}^{i} w_j, \quad i, i \leq d, \right] \]

The sum of products in \( H_{i,d} \) is taken over all possible combinations of \( i \) different \( w_j \)s. Of course, the above expression for \( H_{i,d} \) is a notational compromise, and should not be used to actually calculate \( H_{i,d} \) in a computer program, since it would be very inefficient. It would be like counting the number of cows in a field by first counting the number of legs, and then dividing by four.

Again, the minimum of \( B_d \) occurs when \( w_1 = w_2 = \ldots = w_d = w \). If the buckets are hypercubes, i.e., all \( w \)s are equal, then the expression above reduces to:
\[ B_d = \frac{d}{i=0} K_{r;0} r^d \left( \frac{d}{i!} \right)^i. \]

One must keep in mind that \( B_d \) being the average number of buckets searched for a given search radius \( r \), one must still perform an expected value calculation to obtain the expected number \( \bar{B}_d \) of buckets to search when all values of \( r \) are allowed.

### 3.3 Expected Number of Buckets to Search and Patterns to Compare

For \( k \) \( d \) tree and product partitioning, the expected value \( \bar{B}_d \) for the number of patterns to compare is the product of \( r \) and the expected number of \( d \)-dimensional buckets to search in \( d \)-dimensional space \( \bar{B}_{i,d} \). Calculations for the expected values \( \bar{B}_{i,d} \) and \( \bar{P}_d \) yield:

\[ \bar{B}_{i,d} = E[B_i] = \sum_{i=0}^{d} \left( K_{r;0} \right)^{-1/d} \left( k - 1 \right)! H_{i,d} K_{r;0} \Gamma \left( \frac{i + dk}{d} \right). \]

\[ \bar{P}_d = n \bar{B}_{i,d}. \]

Therefore, \( \bar{B}_d \) and \( \bar{B}_{i,d} \) are independent of \( N \). The larger the total number of patterns, the smaller the buckets, so that the average number of buckets searched is constant. The corresponding value for the case of hypercubic buckets can be obtained from these expressions by making the substitution

\[ H_{i,d} \rightarrow w^{-i} \left( \frac{1}{i!} \right). \]

After simplification, the expected number of buckets for the case of hypercubic buckets is:

\[ \bar{B}_{i,d} = \sum_{i=0}^{d} \left( K_{r;0} \right)^{-1/d} \left( k - 1 \right)! K_{r;0} \Gamma \left( \frac{i + dk}{d} \right). \]

### 3.4 Total Cost

In order to minimize the total cost to retrieve the \( k \)NN, one needs explicit expressions for the total cost as a function of the parameter \( n \). The total cost \( C_t \) is the sum of the overhead cost \( C_o \) and the cost of doing all the distance calculations for all the patterns retrieved:

\[ C_t = C_o + \sum_{i=0}^{d} \bar{P}_d \bar{B}_{i,d}. \]

where \( C_o \) is the cost per pattern of doing a distance calculation without taking the \( p \)th root of the sum. One needs only to calculate the $\sum_{i=1}^{d} x_i - y_i$ to compare distances between patterns. We will now calculate \( C_o \) and \( C_t \) for each of the three algorithms described in Section 2.

#### 3.4.1 Product Partitioning

According to the algorithm of the Appendix, one explicit range calculation must be performed to find the search limits along each slice. The average number of range calculations to be performed to partition a \( d \)-dimensional \( p \)-sphere is the same as the average
number of buckets partitioning a $(d - 1)$-dimensional p-sphere of
the same radius. The overhead cost is the number of these slices,
which is $\overline{B}_{d-1,d}$, times the cost of one explicit range calculation $C_{i,j}$,
plus the cost $C_{b}$ of retrieving the buckets if they are stored on a
slower mass-storage medium

$$C_{b} = \overline{B}_{d-1,d} C_{r} + \overline{B}_{d} C_{b}$$

The total cost $C_{t}$ is plotted in Fig. 1 for the case $C_{b} = 0$ (all patterns
held in active memory), $p = 2$ (Euclidean metric), and $k = 1$,
with costs corresponding to benchmarks that were obtained from a
Digital Vax 9000 computer (0.114 $\mu$s/multiplication or addition,
1.42 $\mu$s/square root extraction). A minimum is seen to exist for $C_{t}$
at a value of $n$ of about 1.8. The value at which this minimum occurs
depends weakly on $d$.

$$C_{t} \approx \overline{B}_{d-1,d} C_{r} + \overline{B}_{d} C_{b}$$

Fig. 1. Total cost $C_{t}$ for $d = 2$ to 10 for the case $C_{b} = 0$ for product
partitioning search, $p = 2$, and $k = 1$, from benchmarks on our computer
(0.114 $\mu$s/multiplication or addition, 1.42 $\mu$s/square root extraction).

3.4.2 Ordered Partitioning

The overhead cost for ordered partitioning comes from the num-
ber of buckets considered, but in $d - 1$ dimensions, since only the
first $d - 1$ dimensions are partitioned in this algorithm. Also, a
search in the ordered partitioning algorithm stops when the upper
and lower siblings of a node fail the distance test, rather than the
parent of the node. This means that up to two additional nodes
must be checked for every group of contiguous nodes passing the
test. At each level $i$, the number of distance calculations is the sum
of the expected number of buckets intersected by the $p$-sphere at
that level, plus twice the number of rows intersected by the
$p$-sphere:

$$\sum_{i=1}^{d-1} (\overline{B}_{i,d} + 2\overline{B}_{i-1,d}) C_{i,j}$$

The cost $C_{i,j}$ of distance calculation depends on the level $i$. An
additional cost occurs when determining the boundaries of the
search within each bucket.

The cost of determining the boundary of the search is simply
that of comparing the value of the $i$th key to that of $D_{max}$. This cost
is minimal, and can be neglected. Therefore, the overhead cost for
ordered partitioning is

$$C_{o} = \sum_{i=1}^{d-1} (\overline{B}_{i,d} + 2\overline{B}_{i-1,d}) C_{i,j} + C_{b} \overline{B}_{d-1,d}$$

Since not all the patterns in each bucket are searched, the expected
number of patterns is different from $n \overline{B}_{i,d}$. The number of pat-
tterns to be searched is proportional to the number of buckets in-
tersected by a $p$-sphere of radius $r$ embedded in a $d$-dimensional
space,

$$\overline{P}_{d} = \sum_{i=1}^{d-1} \left[ \frac{n \overline{B}_{i,d}}{4} \right]$$

while the hypercubic bucket approximation it is

$$\overline{P}_{d} = \sum_{i=1}^{d-1} \frac{K_{p,d}^{-1} d^{-1}}{(k-1)!} H_{d-1} \Gamma \left( \frac{1 + i + dk}{d} \right)$$

In order to compare the performance of the different algorithms studied
in this paper, we calculated the minimum value of $C_{t}$ asso-
ciated with each, for the same cost structure as for Fig. 1. For
that cost structure, we found that there is a significant difference
between algorithms only for small values of $d$. For $d = 2$ ordered
partitioning outperforms the other algorithms studied here by
close to an order of magnitude. This difference decreases rapidly
with $d$ and practically vanishes for $d \geq 5$. The value of $n$ at which
the minimum in $C_{t}$ occurs, which we will call $n_{opt}$, is more or less
independent of $d$, and varies between algorithms. Table I gives
$n_{opt}$ for the same cost structure as that adopted for Fig. 1.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>$n_{opt}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>k-d tree (min)</td>
<td>1.0</td>
</tr>
<tr>
<td>k-d tree (max)</td>
<td>18</td>
</tr>
<tr>
<td>ordered</td>
<td>18</td>
</tr>
<tr>
<td>product</td>
<td>1.7</td>
</tr>
</tbody>
</table>

### 3.4.3 k-d Tree Partitioning

In hierarchical partitioning, the overhead cost is mostly due to the
cost of calculating the range from the test pattern to each node
visited. We want to approximate the average number of nodes
visited $N_{i}$ on level $i$ of the search tree. For a given $\overline{B}_{d,d}$ the
number of nodes visited depends on the distribution of buckets in the
branches of the tree. An upper bound on $N_{i}$ would occur if the
$\overline{B}_{d,d}$ buckets partitioning the search $p$-sphere were distributed as
far from one another as possible. The number of branches would
then be $\overline{B}_{d,d}$, plus $\overline{B}_{d,d}$ nodes to be checked to terminate the
search. There would be the same number of nodes at level $d - 1$
and so on until the number of available nodes became less than
$2 \overline{B}_{d,d}$. The maximum number of nodes at level $i$ is therefore

$$\min(2\overline{B}_{d-1,d}, 2^{i-1})$$

This maximum is very rarely achieved however, since the buckets partitioning the search $p$-sphere tend to lie to-
gether in the k-d tree [11]. The minimum number of nodes is
achieved when all $\overline{B}_{d,d}$ of these buckets are contiguous in the k-d
tree. In this case, the number of nodes visited is

$$\min(N_{i}) = \min \left( \frac{\overline{B}_{d,d}}{2^{i-1} - 1}, 2^{i-1} \right)$$
where \( M = \lfloor \log_2 N \rfloor + 1 \) is the total number of levels in the tree, and the extra node in the first term of the min argument is the minimum cost of terminating the search at each level. \( N_i \) is much closer to \( \min(N) \) than to \( \max(N) \). The overhead cost \( C_o \) is the sum of the distance calculation to the nodes, and the cost of retrieving the buckets that pass the test:

\[
C_o = C_d \sum_{i=1}^{d} N_i + C_{B,\ell}.
\]

### 4 Simulation

We have performed a computer simulation of a NN search based on the product partitioning algorithm described in the Appendix. For each run, 200 instances of stored patterns were generated from a uniform distribution in the \([0, 1]^d\) hypercube. For each set of patterns, the nearest neighbor \((k = 1, p = 2)\) was found for 100 different test patterns from a uniform distribution. The simulation keeps track of the average number of distance calculations, \( \text{avrg}(P_d) \). The only free parameter of the algorithm (apart from \( n \)) is \( r_{\text{inc}} \), the increment factor in the search hypersphere radius. Since \( r_{\text{opt}} \) for this algorithm is of the order of unity, we took \( n = 1 \) for all runs. The expression for \( \bar{P}_d \) in Section 3.3 predicts that \( \text{avrg}(P_d) \) should be equal to 4.0.

Fig. 2 shows \( \text{avrg}(P_d) \) as a function of \( N \), the total number of patterns stored. When \( N \) is small, the presence of boundaries in the hypercube reduces \( \text{avrg}(P_d) \) significantly. That is, when the test pattern is close to a boundary, no bucket needs to be explored in that direction past the boundary. The complexity analysis presented in Section 3 does not take such boundary effects into account. At large values of \( N \), \( \text{avrg}(P_d) \) reaches an asymptotic value slightly larger than 4.0. Part of the explanation of why the asymptotic value of \( \text{avrg}(P_d) \) is larger than 4.0 is that the algorithm, while expanding the search sphere by factors of \( r_{\text{inc}} \), overshoots the range at which the NN is situated. This is why the curve for \( r_{\text{inc}} = 1.1 \) in Fig. 2 is consistently higher than the one for \( r_{\text{inc}} = 1.02 \). In this simulation we found that the effect of increasing \( r_{\text{inc}} \) is not pronounced until \( r_{\text{inc}} > 1.1 \). Of course, there is an overhead cost at keeping track of which buckets have been explored already, and this overhead cost increases rapidly as \( r_{\text{inc}} \) gets closer to 1.0.

![Fig. 2. Average number of patterns searched to find the NN in a simulation, with \( n = 1, k = 1, d = 2 \), as a function of \( N \), for two values of the hypersphere radius increment factor.](image)

The value of \( \text{avrg}(P_d) \) obtained for \( N = 10^6 \) for \( 2 \leq d \leq 10 \) and \( r_{\text{inc}} = 1.02 \) is given in Table 2, along with \( \bar{P}_d \) for \( n = 1 \) for the product partitioning algorithm. Notice that \( \text{avrg}(P_d) \) becomes significantly less than \( P_d \) for dimensions higher than 5 or 6, even with such a large number of patterns. This is because more and more of the buckets become “boundary” buckets as \( d \) increases, and therefore the edge effect becomes more pronounced.

<table>
<thead>
<tr>
<th>( d )</th>
<th>( P_d(n = 1) )</th>
<th>( \text{avrg}(P_d) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4.00</td>
<td>4.14</td>
</tr>
<tr>
<td>3</td>
<td>8.60</td>
<td>8.64</td>
</tr>
<tr>
<td>4</td>
<td>19.0</td>
<td>18.5</td>
</tr>
<tr>
<td>5</td>
<td>42.8</td>
<td>38.0</td>
</tr>
<tr>
<td>6</td>
<td>97.4</td>
<td>75.6</td>
</tr>
<tr>
<td>7</td>
<td>223</td>
<td>139</td>
</tr>
<tr>
<td>8</td>
<td>513</td>
<td>273</td>
</tr>
<tr>
<td>9</td>
<td>1190</td>
<td>461</td>
</tr>
<tr>
<td>10</td>
<td>2750</td>
<td>628</td>
</tr>
</tbody>
</table>

### 5 Discussion and Conclusions

A slight difference exists between the performance of the three algorithms for which total cost was calculated explicitly, especially for low \( d \). This is not to say that one should always choose one algorithm over the others based on these results. The results of the comparison could differ for different values of the costs and the pattern density \( g \) which enters into the calculation of \( C_o \) for the limited ordered partitioning algorithm. For other values of the cost parameters, one can find the optimal bucket size \( r_{\text{opt}} \), by plotting \( C_o \) as a function of \( n \) and finding the minimum, as was done for Fig. 1.

The average cost of the \( k-d \) tree search is expected to be slightly above the minimum calculated, and would therefore not be very different from the cost of the other three algorithms, except at low \( d \). For \( d \leq 3 \), the minimum cost is more than the expected cost for the other algorithms, and the \( k-d \) tree is definitely the least efficient search method among them.

A basic assumption behind the derivations in Section 3 is that the stored patterns are locally uniformly distributed. How are these results affected when the distribution of stored patterns is not uniform, as is most real-world situations? Some authors have tested the robustness of their NNSA and found them to be quite insensitive to the distribution actually used to generate the patterns. For example, [4], [5], [6], and [11] all try several pattern distributions and find only minor differences in \( \text{avrg}(P_d) \) between distributions. In [10] we present a derivation of \( P_d \) for a search sphere centered at the junction between two uniform distributions, with the most abrupt transition possible—a step function. That is, the density of pattern is a constant \( g \) for one half of the search hypersphere, and a different \( g' \) for the other half. But even for that abrupt transition in the distribution of patterns, we show that \( B_2 \) and \( P_2 \) remain the same as if the search disk had been completely within either the \( g' \) or the \( g \) distribution. Given this theoretical demonstration, and the simulations mentioned in the above references, we are fairly confident that the results derived in Section 3 are applicable for most data distributions.

The NNSAs presented here are quite expensive for search spaces of large \( d \). A certain gain in speed over exhaustive search is always expected to occur by the use of a partitioning NNSA, but the gain is not as spectacular for high \( d \) as for low \( d \). For high dimensional search spaces, nonpartitioning NNSA might be more efficient [4],
Moreover, it is not always necessary to find the NN in order to find an acceptable solution to the problem at hand. Often times, a “good” solution is good enough, even though it may not be the best solution, as is the case for artificial neural networks. Similarly, for high dimensionality search spaces, it may often be good enough to pick the closest match within a reasonable number of buckets searched, even though not all the NNs have been searched that would guarantee that the NN is found.

In summary, a complexity analysis of the nearest neighbor search in a continuous d-dimensional space was presented. The expected number of buckets visited and patterns compared were derived for any Minkowski p-metric, whereas only the case p = ∞ had been done before. An optimum value of the bucket size was shown to exist and can be found from the equations for the total cost of the search.

**APPENDIX—ALGORITHMS FOR FAST NN SEARCH**

Let us define r as the radius of the presently-considered search hypersphere, r(q) as its projection along the x₁ axis, and r’ as its projection along the x₂ axis.

**PROCEDURE** FindNN(aₜ)

outputs final winning candidate, Dmin

Initialize r

DO UNTIL found

Dmin = SearchSphere(d, r)

IF Dmin ≤ r THEN

found = TRUE

RETURN Dmin

ELSE

r = r * rinc

ENDIF

SearchSphere(d, r)

; outputs provisional winning candidate, Dmin

FIND imin = index of first slice.

FIND imax = index of last slice.

FIND ipat = index of slice where test pattern is.

FOR i = imin, imax

index(d) = i

IF d ≠ 1 THEN

IF i = ipat, r’ = r

ELSE r’ = center - i

r’ = (r’² + r’0)½

Dmin = MIN(SearchSphere(d-1, r), Dmin)

ELSE

Dmin = MIN(SearchBucket(index))

RETURN winning candidate, Dmin

ENDIF

SearchBucket(index) returns the pattern with the shortest distance to the test pattern found within the bucket with index = index.

**REFERENCES**