A Simple Algorithm for Nearest Neighbor Search in High Dimensions

Kairi Kangro
Institute of Computer Science
University of Tartu
kairikan@ut.ee

ABSTRACT
Finding the closest point in a high-dimensional space is a problem that often occurs in pattern recognition. Unfortunately, the complexity of most known search algorithms grows exponentially with dimension, which makes them unsuitable for high dimensions. However, for most applications, the closest point is of interest only if it is closer than some pre-defined distance. In the article "A Simple Algorithm for Nearest Neighbor Search in High Dimensions" [1], Nene and Nayar introduce an algorithm for finding the closest point within Euclidean distance $\epsilon$, which outperforms other popular search algorithms on a comprehensive set of benchmarks. This essay will give an overview of said article and its results.

1. INTRODUCTION
Finding the nearest neighbor is a common problem in many fields of science and engineering. For multiple dimensions, the problem is stated as follows: given a set of points and a novel query point $Q$, "Find a point in the set such that its distance from $Q$ is lesser than, or equal to, the distance of $Q$ from any other point in the set" [4]. A variety of different search algorithms have been proposed to solve this problem, including $k$-d-trees [2] and $R$-trees [3]. However, the complexity of these algorithms grows exponentially with the dimension of the points, which makes them unsuitable for problems with high ($>15$) dimension. Such high dimensionality appears for example in eigenspace based appearance matching and visual correspondence problems.

Since for most pattern-recognition problems, the closest point is considered a "match" only if it is sufficiently close, the authors of [1] propose an algorithm that finds the nearest neighbor a query point only if it is closed than some pre-specified distance $\epsilon$. Due to this constraint, the complexity of their algorithm does not grow exponentially with the dimensionality, and therefore is able to outperform other known algorithms on search problems with high dimensionality.

The authors also propose a hardware architecture for implementing their algorithm, made feasible by the simplicity of the algorithm. They estimate that a hardware-based implementation would run about 100 times faster than a software-based implementation.

2. THE ALGORITHM
The idea of the algorithm in [1] is as follows: given a set of points $S$ and a query point $Q$, to find the closest point that is within distance $\epsilon$ to $Q$ in $S$, first all points of $S$ within a hypercube with side $2\epsilon$ centered at $Q$ are found, and then an exhaustive search is performed in the cube (See Fig. 1).

![Figure 1: The algorithm finds points inside a hypercube of size $2\epsilon$ centered around the query point $Q$, and then performs an exhaustive search inside the cube. Taken from [1].](image-url)

To find the points in the hypercube, the algorithm does the following. Let $Q_i$, $1 \leq i \leq d$, denote the $i$th coordinate of the point $Q$, where $d$ is the dimensionality of the points. The algorithm first finds the points of $S$ whose first coordinate is between $Q_1 - \epsilon$ and $Q_1 + \epsilon$, then iterates over the remaining dimensions and for every dimension $k$, it discards the points whose $k$th coordinate is not between $Q_k - \epsilon$ and $Q_k + \epsilon$.

To accomplish this, the algorithm uses a preprocessing step, where it creates for each dimension $k$ a sorted array of the $k$th coordinates of the points in $S$ (assumed to be static), and creates a forward and backward map, which map a point to its location in the $k$th sorted array and a coordinate in the sorted array to its corresponding point, respectively. Then the algorithm simply uses two binary searches on each sorted array to determine where the values $Q_k - \epsilon$ and $Q_k + \epsilon$ should be, and from that can easily determine which points have their $k$th coordinate between these two values (See Fig. 2).

It is clear that the running time of the algorithm depends on $\epsilon$, since it determines the number of points chosen for consideration. The authors of [1] show that for $\epsilon$ small enough,
Figure 2: The data structures used by the algorithm. The point set corresponds to the raw data points, and the ordered set consists of \( d \) sorted arrays of the corresponding coordinates. Taken from [1].

The running time of the algorithm on uniformly or normally distributed data is almost independent of the dimensionality (See Fig. 3 and Fig. 4) and grows linearly with respect to the number of points in \( S \). For more details about the complexity of the algorithm and the choice of \( \epsilon \), please see [1].

Figure 3: Average cost of algorithm for uniformly distributed points for dimensions 5, 10, 15, 20, 25. Taken from [1].

Figure 4: Average cost of algorithm for normally distributed points for dimensions 5, 10, 15, 20, 25. Taken from [1].

3. RESULTS

The authors of [1] used a set of benchmarks to compare their algorithm against other popular search algorithms, including \( k \)-\( d \)-trees and R-trees. They used both structured test data, where the given point set had either uniform or normal distribution, and unstructured data, whose distribution was not known. Both types of benchmarks were run with varying dimensionality and number of points. The proposed algorithm outperformed all of the other algorithms used, although for uniformly distributed data, it performed only slightly better than exhaustive search. For more details about the benchmarks, see [1].

The authors also gave two real-life applications of their algorithm, namely real time object recognition and motion estimation for MPEG coding. The first uses a data set of 36,000 points with 35 dimensions, and the second uses 961 points from a 256-dimensional space. In both cases, the performance of the proposed algorithms was compared to other algorithms commonly used in these applications. Table 1 shows the average running time of different algorithms on object recognition, taken over 100,000 closest point searches. Table 2 shows the average execution time per frame on MPEG motion vector estimation of the proposed algorithm and exhaustive search (commonly used in MPEG coding [1]). For both applications, the proposed algorithm outperforms the other algorithms.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proposed algorithm</td>
<td>.0025</td>
</tr>
<tr>
<td>( k )-( d )-tree</td>
<td>.0045</td>
</tr>
<tr>
<td>Exhaustive Search</td>
<td>.1533</td>
</tr>
<tr>
<td>Projection Search</td>
<td>.2924</td>
</tr>
</tbody>
</table>

Table 2: Average execution time per frame for MPEG motion estimation. Taken from [1]

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proposed algorithm</td>
<td>1.5</td>
</tr>
<tr>
<td>Correlation(SSD)</td>
<td>11</td>
</tr>
</tbody>
</table>

Since the main computations performed by the algorithm are simple integer map lookups and integer comparisons, the authors claim that it is possible to implement this algorithm in hardware using inexpensive components, and estimate a 100-fold speedup of such an implementation over any software implementation.

4. CONCLUSIONS

The algorithm introduced by Nene and Nayar in [1] for solving a constrained version of the nearest neighbor problem outperforms other commonly used algorithms in high dimensions both on benchmarks and in two real-life applications. The possibility of implementing this algorithm inexpensively in hardware for further speedup makes this an attractive algorithm for any kind of research that deals with high-dimensional nearest neighbor search.

5. REFERENCES