An Optimal Algorithm for Approximate Nearest Neighbor Searching in Fixed Dimensions*

Sunil Arya † David M. Mount‡ Nathan S. Netanyahu§ Ruth Silverman¶
Angela Y. Wu||

July 6, 1998

Abstract

Consider a set S of n data points in real d-dimensional space, R^d, where distances are measured using any Minkowski metric. In nearest neighbor searching we preprocess S into a data structure, so that given any query point $q \in R^d$, the closest point of S to q can be reported quickly. Given any positive real ϵ, a data point p is a $(1+\epsilon)$-*approximate nearest neighbor* of q if its distance from q is within a factor of $(1+\epsilon)$ of the distance to the true nearest neighbor. We show that it is possible to preprocess a set of n points in R^d in $O(dn \log n)$ time and $O(dn)$ space, so that given a query point $q \in R^d$, and $\epsilon > 0$, a $(1+\epsilon)$-approximate nearest neighbor of q can be computed in $O(c_{d,\epsilon} \log n)$ time, where $c_{d,\epsilon} \leq d \lceil 1 + 6d/\epsilon \rceil^d$ is a factor depending only on dimension and ϵ. In general, we show that given an integer $k \geq 1$, $(1+\epsilon)$-approximations to the k nearest neighbors of q can be computed in additional $O(kd \log n)$ time.
Key words: Nearest neighbor searching, post-office problem, closest-point queries, approximation algorithms, box-decomposition trees, priority search.

1 Introduction.

Nearest neighbor searching is the following problem: we are given a set S of n *data points* in a metric space, X, and the task is to preprocess these points so that, given any *query point* $q \in X$, the data point nearest to q can be reported quickly. This is also called the *closest-point problem*

*A preliminary version of this paper appeared in the *Proc. of the Fifth Annual ACM-SIAM Symp. on Discrete Algorithms*, 1994, pp. 573–582.

†Department of Computer Science, The Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong. The work of this author was partially supported by HK RGC grant HKUST 736/96E. Part of this research was conducted while the author was visiting the Max-Planck-Institut für Informatik, Saarbrücken, Germany. Email: arya@cs.ust.hk

‡Department of Computer Science and Institute for Advanced Computer Studies, University of Maryland, College Park, Maryland. The support of the National Science Foundation under grant CCR–9712379 is gratefully acknowledged. Email: mount@cs.umd.edu

§Center for Automation Research, University of Maryland, College Park, Maryland, and Space Data and Computing Division, NASA Goddard Space Flight Center, Greenbelt, Maryland. This research was carried out, in part, while the author held a National Research Council NASA Goddard Associateship. Email: nathan@cfar.umd.edu

¶Department of Computer Science, University of the District of Columbia, Washington, DC, and Center for Automation Research, University of Maryland, College Park, Maryland. The support of the National Science Foundation under grant CCR–9310705 is gratefully acknowledged. Email: ruth@cfar.umd.edu

||Department of Computer Science and Information Systems, The American University, Washington, DC. Email: awu@american.edu

DOCKET ALARM
Find authenticated court documents without watermarks at docketalarm.com.

and the *post office problem*. Nearest neighbor searching is an important problem in a variety of applications, including knowledge discovery and data mining [FPSSU96], pattern recognition and classification [CH67, DH73], machine learning [CS93], data compression [GG91], multimedia databases [FSN+95], document retrieval [DDF+90], and statistics [DW82].

High-dimensional nearest neighbor problems arise naturally when complex objects are represented by vectors of d numeric features. Throughout we will assume the metric space X is real d-dimensional space R^d. We also assume distances are measured using any Minkowski L_m distance metric. For any integer $m \geq 1$, the L_m-*distance* between points $p = (p_1, p_2, \ldots, p_d)$ and $q = (q_1, q_2, \ldots, q_d)$ in R^d is defined to be the m-th root of $\sum_{1 \leq i \leq d} |p_i - q_i|^m$. In the limiting case, where $m = \infty$, this is equivalent to $\max_{1 \leq i \leq d} |p_i - q_i|$. The L_1, L_2, and L_∞ metrics are the well-known Manhattan, Euclidean and max metrics, respectively. We assume that the distance between any two points in R^d can be computed in $O(d)$ time. (Note that the root need not be computed when comparing distances.) Although this framework is strong enough to include many nearest neighbor applications, it should be noted that there are applications that do not fit within this framework (e.g., computing the nearest neighbor among strings, where the distance function is the edit distance, the number of single character changes).

Obviously the problem can be solved in $O(dn)$ time through simple brute-force search. A number of methods have been proposed which provide relatively modest constant factor improvements (e.g., through partial distance computation [BG85], or by projecting points onto a single line [FBS75, GK92, LC94]). Our focus here is on methods using data structures that are stored in main memory. There is a considerable literature on nearest neighbor searching in databases. For example, see [BBKK97, BKK96, LJF94, RKV95, WJ96].

For uniformly distributed point sets, good expected case performance can be achieved by algorithms based on simple decompositions of space into regular grids. Rivest [Riv74] and later Cleary [Cle79] provided analyses of these methods. Bentley, Weide, and Yao [BWY80] also analyzed a grid-based method for distributions satisfying certain bounded-density assumptions. These results were generalized by Friedman, Bentley, and Finkel [FBF77] who showed that $O(n)$ space and $O(\log n)$ query time are achievable in the expected case through the use of kd-trees. However, even these methods suffer as dimension increases. The constant factors hidden in the asymptotic running time grow at least as fast as 2^d (depending on the metric). Sproull [Spr91] observed that the empirically measured running time of kd-trees does increase quite rapidly with dimension. Arya, et al. [AMN95] showed that if n is not significantly larger than 2^d, as arises in some applications, then boundary effects mildly decrease this exponential dimensional dependence.

From the perspective of worst-case performance, an ideal solution would be to preprocess the points in $O(n \log n)$ time, into a data structure requiring $O(n)$ space so that queries can be answered in $O(\log n)$ time. In dimension 1 this is possible by sorting the points, and then using binary search to answer queries. In dimension 2, this is also possible by computing the Voronoi diagram for the point set and then using any fast planar point location algorithm to locate the cell containing the query point. (For example, see [dBvKOS97, Ede87, PS85].) However, in dimensions larger than 2, the worst-case complexity of the Voronoi diagram grows as $O(n^{\lceil d/2 \rceil})$. Higher dimensional solutions with sublinear worst-case performance were considered by Yao and Yao [YY85]. Later Clarkson [Cla88] showed that queries could be answered in $O(\log n)$ time with $O(n^{\lceil d/2 \rceil + \delta})$ space, for any $\delta > 0$. The O-notation hides constant factors that are exponential in d. Agarwal and Matoušek [AM93a] generalized this by providing a tradeoff between space and query time. Meiser [Mei93] showed that exponential factors in query time could be eliminated by giving an algorithm with $O(d^5 \log n)$ query time and $O(n^{d+\delta})$ space, for any $\delta > 0$. In any fixed dimension greater than 2, no known method achieves the simultaneous goals of roughly linear space and logarithmic query

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

time.

The apparent difficulty of obtaining algorithms that are efficient in the worst case with respect to both space and query time for dimensions higher than 2, suggests that the alternative approach of finding *approximate* nearest neighbors is worth considering. Consider a set S of data points in R^d and a query point $q \in R^d$. Given $\epsilon > 0$, we say that a point $p \in S$ is a $(1+\epsilon)$-*approximate nearest neighbor* of q if

$$dist(p,q) \leq (1+\epsilon)\, dist(p^*,q),$$

where p^* is the true nearest neighbor to q. In other words, p is within relative error ϵ of the true nearest neighbor. More generally, for $1 \leq k \leq n$, a kth $(1+\epsilon)$-approximate nearest neighbor of q is a data point whose relative error from the true kth nearest neighbor of q is ϵ. For $1 \leq k \leq n$, define a *sequence* of k approximate nearest neighbors of query point q to be a sequence of k distinct data points, such that the ith point in the sequence is an approximation to the ith nearest neighbor of q. Throughout we assume that both d and ϵ are fixed constants, independent of n, but we will include them in some of the asymptotic results to indicate the dependency on these values.

The approximate nearest neighbor problem has been considered by Bern [Ber93]. He proposed a data structure based on quadtrees, which uses linear space and provides logarithmic query time. However, the approximation error factor for his algorithm is a fixed function of the dimension. Arya and Mount [AM93c] proposed a randomized data structure which achieves polylogarithmic query time in the expected case, and nearly linear space. In their algorithm the approximation error factor ϵ is an arbitrary positive constant, fixed at preprocessing time. In this paper, we strengthen these results significantly. Our main result is stated in the following theorem.

Theorem 1 *Consider a set S of n data points in R^d. There is a constant $c_{d,\epsilon} \leq d\lceil 1+6d/\epsilon \rceil^d$, such that in $O(dn\log n)$ time it is possible to construct a data structure of size $O(dn)$, such that for any Minkowski metric:*

- *(i) Given any $\epsilon > 0$ and $q \in R^d$, a $(1+\epsilon)$-approximate nearest neighbor of q in S can be reported in $O(c_{d,\epsilon}\log n)$ time.*
- *(ii) More generally, given $\epsilon > 0$, $q \in R^d$, and any k, $1 \leq k \leq n$, a sequence of k $(1+\epsilon)$-approximate nearest neighbors of q in S can be computed in $O((c_{d,\epsilon}+kd)\log n)$ time.*

In the case of a single nearest neighbor and for fixed d and ϵ, the space and query times given in Theorem 1 are asymptotically optimal in the algebraic decision tree model of computation. This is because $O(n)$ space and $O(\log n)$ time are required to distinguish between the n possible outcomes in which the query point coincides with one of the data points. We make no claims of optimality for the factor $c_{d,\epsilon}$.

Recently there have been a number of results showing that with significantly more storage, it is possible to improve the dimensional dependencies in query time. Clarkson [Cla94] showed that query time could be reduced to $O((1/\epsilon)^{d/2}\log n)$ with $O((1/\epsilon)^{d/2}(\log\rho)n)$ space, where ρ is the ratio between the furthest-pair and closest-pair interpoint distances. Later Chan [Cha97] showed that the factor of $\log\rho$ could be removed from the space complexity. Kleinberg [Kle97] showed that it is possible to eliminate exponential dependencies on dimension in query time, but with $O(n\log d)^{2d}$ space. Recently, Indyk and Motwani [IM98] and independently Kushilevitz, Ostrovsky and Rabani [KOR98], have announced algorithms that eliminate all exponential dependencies in dimension, yielding a query time $O(d\ \log^{O(1)}(dn))$ and space $(dn)^{O(1)}$. Here the O-notation hides constant factors depending exponentially on ϵ, but not on dimension.

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

There are two important practical aspects of Theorem 1. First, space requirements are completely independent of ϵ and are asymptotically optimal for all parameter settings, since dn storage is needed just to store the data points. In applications where n is large and ϵ is small, this is an important consideration. Second, preprocessing is independent of ϵ and the metric, implying that once the data structure has been built, queries can be answered for any error bound ϵ and for any Minkowski metric. In contrast, all the above mentioned methods would require that the data structure be rebuilt if ϵ or the metric changes. In fact, setting $\epsilon = 0$ will cause our algorithm to compute the true nearest neighbor, but we cannot provide bounds on running time, other than a trivial $O(dn \log n)$ time bound needed to search the entire tree by our search algorithm. Unfortunately, exponential factors in query time do imply that our algorithm is not practical for large values of d. However, our empirical evidence in Section 6 shows that the constant factors are much smaller than the bound given in Theorem 1 for the many distributions that we have tested. Our algorithm can provide significant improvements over brute-force search in dimensions as high as 20, with a relatively small average error. There are a number of important applications of nearest neighbor searching in this range of dimensions.

The algorithms for both preprocessing and queries are deterministic and easy to implement. Our data structure is based on a hierarchical decomposition of space, which we call a *balanced box-decomposition (BBD) tree*. This tree has $O(\log n)$ height, and subdivides space into regions of $O(d)$ complexity defined by axis-aligned hyperrectangles that are *fat*, meaning that the ratio between the longest and shortest sides is bounded. This data structure is similar to balanced structures based on box-decomposition [BET93, CK95, Bes95], but there are a few new elements that have been included for the purposes of nearest neighbor searching and practical efficiency. Space is recursively subdivided into a collection of *cells*, each of which is either a d-dimensional rectangle or the set-theoretic difference of two rectangles, one enclosed within the other. Each node of the tree is associated with a cell, and hence it is implicitly associated with the set of data points lying within this cell. Each leaf cell is associated with a single point lying within the bounding rectangle for the cell. The leaves of the tree define a subdivision of space. The tree has $O(n)$ nodes and can be built in $O(dn \log n)$ time.

Here is an intuitive overview of the approximate nearest neighbor query algorithm. Given the query point q, we begin by locating the leaf cell containing the query point in $O(\log n)$ time by a simple descent through the tree. Next, we begin enumerating the leaf cells in increasing order of distance from the query point. We call this *priority search*. When a cell is visited, the distance from q to the point associated with this cell is computed. We keep track of the closest point seen so far. For example, Figure 1(a) shows the cells of such a subdivision. Each cell has been numbered according to its distance from the query point.

Let p denote the closest point seen so far. As soon as the distance from q to the current leaf cell exceeds $dist(q, p)/(1 + \epsilon)$ (illustrated by the dotted circle in Figure 1(a)), it follows that the search can be terminated, and p can be reported as an approximate nearest neighbor to q. The reason is that any point located in a subsequently visited cell cannot be close enough to q to violate p's claim to be an approximate nearest neighbor. (In the example shown in the figure, the search terminates just prior to visiting cell 9. In this case p is not the true nearest neighbor, since that point belongs to cell 9, which was never visited.) We will show that, by using an auxiliary heap, priority search can be performed in time $O(d \log n)$ times the number of leaf cells that are visited.

We will also show that the number of cells visited in the search depends on d and ϵ, but not on n. Here is an intuitive explanation (and details will be given in Lemma 5). Consider the last leaf cell to be visited that did not cause the algorithm to terminate. If we let r denote the distance from q to this cell, and let p denote the closest data point encountered so far, then because we do

DOCKET
ALARM
Find authenticated court documents without watermarks at docketalarm.com.

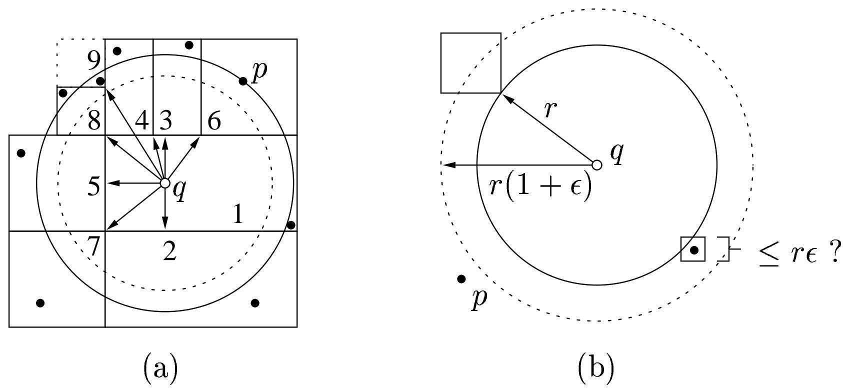

Figure 1: Algorithm overview.

not terminate, we know that the distance from q to p is at least $r(1+\epsilon)$. (See Figure 1(b).) We could not have seen a leaf cell of diameter less than $r\epsilon$ up to now, since the associated data point would necessarily be closer to q than p. This provides a lower bound on the sizes of the leaf cells seen. The fact that cells are fat and a simple packing argument provide an upper bound on the number of cells encountered.

It is an easy matter to extend this algorithm to enumerate data points in "approximately" increasing distance from the query point. In particular we will show that a simple generalization to this search strategy allows us to enumerate a sequence of k approximate nearest neighbors of q in additional $O(kd \log n)$ time. We will also show that, as a consequence of the results of Callahan and Kosaraju [CK95] and Bespamyatnikh [Bes95], the data structure can be generalized to handle point insertions and deletions in $O(\log n)$ time per update.

The rest of the paper is organized as follows. In Section 2 we introduce the BBD-tree data structure, present an algorithm for its construction, and analyze its structure. In Section 3 we establish the essential properties of the BBD-tree which are used for the nearest neighbor algorithm. In Section 4 we present the query algorithm for the nearest neighbor problem, and in Section 5 we present its generalization to enumerating the k approximate nearest neighbors. In Section 6 we present experimental results.

2 The BBD-tree.

In this section we introduce the *balanced box-decomposition tree* or *BBD-tree*, which is the primary data structure used in our algorithm. It is among the general class of geometric data structures based on a hierarchical decomposition of space into d-dimensional rectangles whose sides are orthogonal to the coordinate axes. The main feature of the BBD-tree is that it combines in one data structure two important features that are present in these data structures.

First consider the optimized kd-tree [FBF77]. This data structure recursively subdivides space by a hyperplane that is orthogonal to one of the coordinate axes and which partitions the data points as evenly as possible. As a consequence, as one descends any path in the tree the *cardinality* of points associated with the nodes on this path decreases exponentially. In contrast, consider quadtree-based data structures, which decompose space into regions that are either hypercubes, or generally rectangles whose *aspect ratio* (the ratio of the length of the longest side to the shortest side) is bounded by a constant. These include PR-quadtrees (see Samet [Sam90]), and structures by Clarkson [Cla83], Feder and Greene [FG88], Vaidya [Vai89], Callahan and Kosaraju [CK92],

DOCKET ALARM
Find authenticated court documents without watermarks at docketalarm.com.

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS

Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS

Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS

Sync your system to PACER to automate legal marketing.