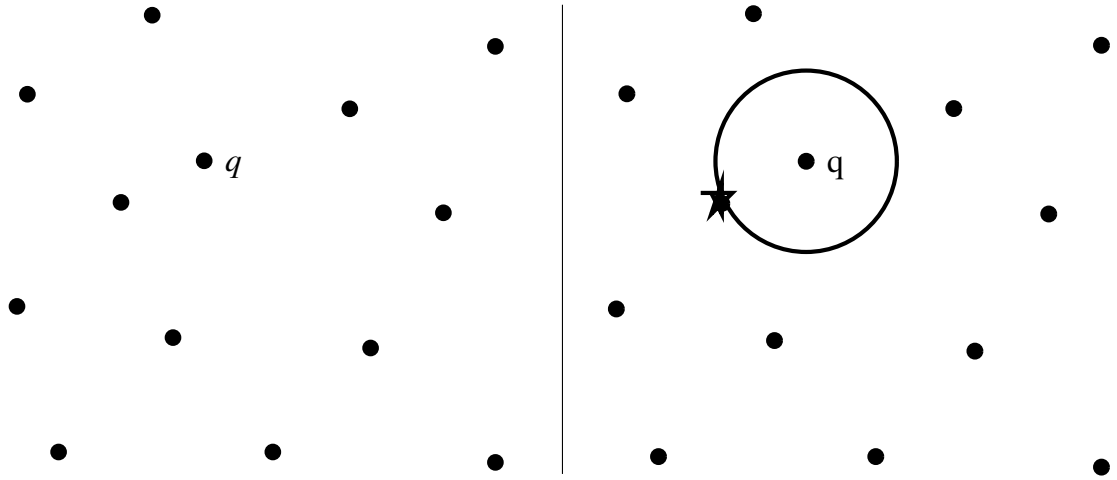

INPUT OUTPUT

17.5 Nearest Neighbor Search

Input description: A set S of n points in d dimensions; a query point q.

Problem description: Which point in S is closest to q?

Discussion: The need to quickly find the nearest neighbor of a query point arises in a variety of geometric applications. The classic example involves designing a system to dispatch emergency vehicles to the scene of a fire. Once the dispatcher learns the location of the fire, she uses a map to find the firehouse closest to this point to minimize transportation delays. This situation occurs in any application mapping customers to service providers.

A nearest-neighbor search is also important in classification. Suppose we are given a collection of numerical data about people (say age, height, weight, years of education, and income level) each of whom has been labeled as Democrat or Republican. We seek a classifier to decide which way a given person is likely to vote. Each of the people in our data set is represented by a party-labeled point in d-dimensional space. A simple classifier can be built by assigning the new point to the party affiliation of its nearest neighbor.

Such nearest-neighbor classifiers are widely used, often in high-dimensional spaces. The vector-quantization method of image compression partitions an image into 8×8 pixel regions. This method uses a predetermined library of several thousand 8×8 pixel tiles and replaces each image region by the most similar library tile. The most similar tile is the point in 64-dimensional space that is closest to

the image region in question. Compression is achieved by reporting the identifier of the closest library tile instead of the 64 pixels, at some loss of image fidelity.

Issues arising in nearest-neighbor search include:

- *How many points are you searching?* – When your data set contains only a small number of points (say $n \leq 100$), or if only a few queries are ever destined to be performed, the simple approach is best. Compare the query point q against each of the n data points. Only when fast queries are needed for large numbers of points does it pay to consider more sophisticated methods.

- *How many dimensions are you working in?* – A nearest neighbor search gets progressively harder as the dimensionality increases. The *kd*-tree data structure, presented in Section 12.6 (page 389), does a very good job in moderate-dimensional spaces—even the plane. Still, above 20 dimensions or so, kd-tree search degenerates pretty much to a linear search through the data points. Searches in high-dimensional spaces become hard because a sphere of radius r, representing all the points with distance $\leq r$ from the center, progressively fills up less volume relative to a cube as the dimensionality increases. Thus, any data structure based on partitioning points into enclosing volumes will become progressively less effective.

  In two dimensions, Voronoi diagrams (see Section 17.4 (page 576)) provide an efficient data structure for nearest-neighbor queries. The Voronoi diagram of a point set in the plane decomposes the plane into regions such that the cell containing data point p consists of all points that are nearer to p than any other point in S. Finding the nearest neighbor of query point q reduces to finding which Voronoi diagram cell contains q and reporting the data point associated with it. Although Voronoi diagrams can be built in higher dimensions, their size rapidly grows to the point of unusability.

- *Do you really need the exact nearest neighbor?* – Finding the exact nearest neighbor in a very high dimensional space is hard work; indeed, you probably won't do better than a linear (brute force) search. However, there are algorithms/heuristics that can give you a reasonably close neighbor of your query point fairly quickly.

  One important technique is *dimension reduction*. Projections exist that map any set of n points in d-dimensions into a $d' = O(\lg n/\epsilon^2)$-dimensional space such that distance to the nearest neighbor in the low-dimensional space is within $(1+\epsilon)$ times that of the actual nearest neighbor. Projecting the points onto a random hyperplane of dimension d' in E^d will likely do the trick.

  Another idea is adding some randomness when you search your data structure. A *kd*-tree can be efficiently searched for the cell containing the query point q—a cell whose boundary points are good candidates to be close neighbors. Now suppose we search for a point q', which is a small random perturbations of q. It should land in a different but nearby cell, one of whose

boundary points might prove to be an even closer neighbor of q. Repeating such random queries gives us a way to productively use exactly as much computing time as we are willing to spend to improve the answer.

- *Is your data set static or dynamic?* – Will there be occasional insertions or deletions of new data points in your application? If these are very rare events, it might pay to rebuild your data structure from scratch each time. If they are frequent, select a version of the kd-tree that supports insertions and deletions.

The nearest neighbor graph on a set S of n points links each vertex to its nearest neighbor. This graph is a subgraph of the Delaunay triangulation, and so can be computed in $O(n \log n)$. This is quite a bargain, since it takes $\Theta(n \log n)$ time just to discover the closest pair of points in S.

As a lower bound, the closest-pair problem reduces to sorting in one dimension. In a sorted set of numbers, the closest pair corresponds to two numbers that lie next to each other, so we need only check that which is the minimum gap between the $n - 1$ adjacent pairs. The limiting case occurs when the closest pair are zero distance apart, meaning that the elements are not unique.

Implementations: *ANN* is a C++ library for both exact and approximate nearest neighbor searching in arbitrarily high dimensions. It performs well for searches over hundreds of thousands of points in up to about 20 dimensions. It supports all l_p distance norms, including Euclidean and Manhattan distance, and is available at *http://www.cs.umd.edu/~mount/ANN/*. It is the first code I would turn to for nearest neighbor search.

Samet's spatial index demos (*http://donar.umiacs.umd.edu/quadtree/*) provide a series of Java applets illustrating many variants of *kd*-trees, in association with [Sam06]. *KDTREE 2* contains C++ and Fortran 95 implementations of *kd*-trees for efficient nearest nearest neighbor search in many dimensions. See *http://arxiv.org/abs/physics/0408067*.

Ranger [MS93] is a tool for visualizing and experimenting with nearest-neighbor and orthogonal-range queries in high-dimensional data sets, using multidimensional search trees. Four different search data structures are supported by *Ranger*: naive kd-trees, median kd-trees, nonorthogonal kd-trees, and the vantage point tree. It is available in the algorithm repository *http://www.cs.sunysb.edu/~algorith*.

Nearpt3 is a special-purpose code for nearest-neighbor search of extremely large point sets in three dimensions. See *http://wrfranklin.org/Research/nearpt3*.

See Section 17.4 (page 576) for a complete collection of Voronoi diagram implementations. In particular, CGAL (*www.cgal.org*) and LEDA (see Section 19.1.1 (page 658)) provide implementations of Voronoi diagrams in C++, as well as planar point location to make effective use of them for nearest-neighbor search.

Notes: Indyk [Ind04] ably surveys recent results in approximate nearest neighbor search in high dimensions based on random projection methods. Both theoretical [IM04] and empirical [BM01] results indicate that the method preserves distances quite nicely.

The theoretical guarantees underlying Ayra and Mount's approximate nearest neighbor code *ANN* [AM93, AMN+98] are somewhat different. A sparse weighted graph structure is built from the data set, and the nearest neighbor is found by starting at a random point and greedily walking towards the query point in the graph. The closest point found over several random trials is declared the winner. Similar data structures hold promise for other problems in high-dimensional spaces. Nearest-neighbor search was a subject of the Fifth DIMACS challenge, as reported in [GJM02].

Samet [Sam06] is the best reference on kd-trees and other spatial data structures. All major (and many minor) variants are developed in substantial detail. A shorter survey [Sam05] is also available. The technique of using random perturbations of the query point is due to [Pan06].

Good expositions on finding the closest pair of points in the plane [BS76] include [CLRS01, Man89]. These algorithms use a divide-and-conquer approach instead of just selecting from the Delaunay triangulation.

Related Problems: Kd-trees (see page 389), Voronoi diagrams (see page 576), range search (see page 584).