

# SANDIA REPORT

SAND2010-4313  
Unlimited Release  
Printed July 2010

## Efficient Nearest Neighbor Searches in N-ABLE™

Greg E. Mackey

Prepared by  
Sandia National Laboratories  
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Approved for public release; further dissemination unlimited.



**Sandia National Laboratories**

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

**NOTICE:** This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from  
U.S. Department of Energy  
Office of Scientific and Technical Information  
P.O. Box 62  
Oak Ridge, TN 37831

Telephone: (865) 576-8401  
Facsimile: (865) 576-5728  
E-Mail: [reports@adonis.osti.gov](mailto:reports@adonis.osti.gov)  
Online ordering: <http://www.osti.gov/bridge>

Available to the public from  
U.S. Department of Commerce  
National Technical Information Service  
5285 Port Royal Rd  
Springfield, VA 22161

Telephone: (800) 553-6847  
Facsimile: (703) 605-6900  
E-Mail: [orders@ntis.fedworld.gov](mailto:orders@ntis.fedworld.gov)  
Online ordering: <http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online>



# Efficient Nearest Neighbor Searches in N-ABLE<sup>TM</sup>

Greg E. Mackey  
Computational Economics Group  
Sandia National Laboratories  
P.O. Box 5800  
Albuquerque, NM 87185  
gemacke@sandia.gov

## Abstract

The nearest neighbor search is a significant problem in transportation modeling and simulation. This paper describes how the nearest neighbor search is implemented efficiently with respect to running time in the NISAC Agent-Based Laboratory for Economics. The paper shows two methods to optimize running time of the nearest neighbor search. The first optimization uses a different distance metric that is more computationally efficient. The concept of a magnitude-comparable distance is described, and the paper gives a specific magnitude-comparable distance that is more computationally efficient than the actual distance function. The paper also shows how the given magnitude-comparable distance can be used to speed up the actual distance calculation. The second optimization reduces the number of points the search examines by using a spatial data structure. The paper concludes with testing of the different techniques discussed and the results.



# 1 Introduction

The nearest neighbor search is a significant problem in transportation modeling and simulation. Transportation infrastructures are naturally represented by graphs. Often, queries against the graph are not restricted to the vertices of the graph but are allowed for any points in the continuous space in which the graph exists. The nearest neighbor search is one method to translate from the continuous space to the discrete set of vertices of the graph. The nearest neighbor search can be defined as given a set of points  $P$  in the continuous space  $S$  and a point  $q \in S$ , find the point in  $P$  that is nearest to  $q$ .

This paper discusses the nearest neighbor search and describes how it is implemented efficiently with respect to running time in the NISAC Agent-Based Laboratory for Economics (N-ABLE<sup>TM</sup>) [3]. There are two main optimizations for the nearest neighbor search. The first optimization is to use a more computationally efficient distance metric because the distance calculation is repeated numerous times in the search. The exact distance is the distance between two points on the earth which can be approximated as a great-circle distance. A great-circle distance is the shortest distance between any two points on the surface of a sphere measured along a path on the surface of the sphere. Unfortunately, the great-circle distance calculation is somewhat expensive. A magnitude-comparable distance can be a less expensive substitute. The second optimization is to reduce the number of points examined by the search. The simple approach is a linear search, but using a spatial data structure can significantly decrease the number of points examined by the search.

This paper begins by describing the problem of calculating the distance between two points on the earth and discusses using the haversine formula to calculate great-circle distance in Section 2. In Section 3 the concept of a magnitude-comparable distance, which is a calculated distance that is often less expensive and can be used when the result of the comparison of the magnitudes of distances is more important than the distances themselves, is described. The author also explains how to use it to speed up the nearest neighbor search. Section 4 describes how to use a magnitude-comparable distance to speed up the great-circle distance calculation. Section 5 discusses various spatial data structures that can speed up the nearest neighbor search. Section 6 discusses how N-ABLE<sup>TM</sup> performs distance calculations and uses the nearest neighbor search. Section 7 gives the testing and results. Finally, Section 8 gives some concluding thoughts, and Section 9 gives some acknowledgments.

## 2 Calculating the Distance between Two Points on the Earth

One excellent source that discusses finding the distance between two points on the earth is Section 5.1 of the Geographic Information Systems FAQ [2]. Assuming that the earth can be represented as a perfect sphere, the problem can be represented as finding the great-circle distance between two points on a sphere. The assumption introduces a small amount of error into the calculation, but for the author's purposes the error is acceptable.

Let  $\phi$  refer to latitude and  $\lambda$  refer to longitude. All latitudes and longitudes are assumed to be in radians. The goal is to find the distance between the two points  $p_1 = (\phi_1, \lambda_1)$  and  $p_2 = (\phi_2, \lambda_2)$ . The latitudinal difference is  $\Delta\phi = \phi_2 - \phi_1$ , and the longitudinal difference is  $\Delta\lambda = \lambda_2 - \lambda_1$ . Let  $s$  be the great-circle distance, or arc length, and let  $\theta$  be the central angle. Also, let  $r$  be the radius.



**Figure 1.** Great-circle ( $s$ ) and tunnel-through ( $d$ ) distances.

Figure 1 shows the geometry of the problem.

The general formula for arc length is

$$s = r\theta.$$

Since finding the arc length is a simple calculation involving the central angle, the author focuses on finding the central angle. One way to calculate the central angle is to use the spherical law of cosines:

$$\theta = \arccos(\sin \phi_1 \sin \phi_2 + \cos \phi_1 \cos \phi_2 \cos \Delta\lambda).$$

This method can have large rounding errors when the points are close to each other, so it is normally not used. The preferred method is the haversine formula [6]:

$$\theta = 2 \arcsin \left( \sqrt{\sin^2 \left( \frac{\Delta\phi}{2} \right) + \cos \phi_1 \cos \phi_2 \sin^2 \left( \frac{\Delta\lambda}{2} \right)} \right).$$

This formula is accurate for all cases except where the points are near antipodal, or on opposite ends of the earth. When this happens rounding errors can make the parameter of arcsine greater than one for which arcsine is undefined. To protect against this case, a min function is introduced around the square root to ensure the value passed to arcsine is not greater than 1. Expressing in equation form and plugging into the general equation for arc length yields the formula:

$$s = 2r \arcsin \left( \min \left[ 1, \sqrt{\sin^2 \left( \frac{\Delta\phi}{2} \right) + \cos \phi_1 \cos \phi_2 \sin^2 \left( \frac{\Delta\lambda}{2} \right)} \right] \right).$$

Listing 1 shows C++ code that performs this computation efficiently.

### 3 Optimizing the Nearest Neighbor Search's Distance Calculation

The natural distance metric for performing a nearest neighbor search for points on a globe is great-circle distance, which can be calculated using the haversine formula. However, the haversine formula is somewhat expensive. The nearest neighbor search would perform faster if the distance calculation required less work to calculate.

One potentially less expensive distance calculation is a magnitude-comparable distance. A magnitude-comparable distance guarantees that its comparison to another magnitude-comparable

```

double greatCircleDistance(const Point& p1, const Point& p2)
{
    // 2 * average radius of earth (in miles)
    const double radius2 = 7912;

    double sinLatDiff = sin((p2.lat - p1.lat) * .5);
    double sinLonDiff = sin((p2.lon - p1.lon) * .5);

    double a = sqrt(sinLatDiff * sinLatDiff +
                   cos(p1.lat) * cos(p2.lat) *
                   sinLonDiff * sinLonDiff);

    if (a > 1) a = 1;

    return radius2 * asin(a);
}

```

**Listing 1.** Great-circle distance using haversine formula.

distance will always yield the same relative result as the comparison of the two actual distances the magnitude-comparable distances replace. Let  $n_1$  be the actual distance between the two points  $p_1$  and  $p_2$ , and let  $m_1$  be some other distance metric between the same two points. Likewise, define  $n_2$  and  $m_2$  for the points  $p_3$  and  $p_4$ . The distance  $m$  is a magnitude-comparable distance for  $n$  if and only if the following are true:

$$\begin{aligned}
 n_1 < n_2 &\leftrightarrow m_1 < m_2 \\
 n_1 = n_2 &\leftrightarrow m_1 = m_2 \\
 n_1 > n_2 &\leftrightarrow m_1 > m_2.
 \end{aligned}$$

The nearest neighbor search is one such problem where a magnitude-comparable distance can be substituted for the actual distance since the relative ordering of the distances is more important than the actual distances.

One candidate for a magnitude-comparable distance is the “tunnel-through” distance, the chord of the great-circle arc connecting the two points. This distance, shown as  $d$  in Figure 1, is inexpensive to calculate if the Cartesian coordinates of the points are known. With the Cartesian points  $p_1 = (x_1, y_1, z_1)$  and  $p_2 = (x_2, y_2, z_2)$ , the tunnel-through distance can be calculated using the standard Cartesian distance formula:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}.$$

An even more inexpensive distance metric can be achieved by using the square of the tunnel-through distance,  $d^2$ , which avoids the cost of the square root operation. This method requires only eight floating point operations while the haversine formula requires ten floating point operations, one square root operation, and five trigonometric operations. Listing 2 shows C++ code that performs the computation efficiently.

```

double comparableDistance(const Point& p1, const Point& p2)
{
    double xdiff = p2.x - p1.x;
    double ydiff = p2.y - p1.y;
    double zdiff = p2.z - p1.z;

    return xdiff * xdiff + ydiff * ydiff + zdiff * zdiff;
}

```

**Listing 2.** Magnitude-comparable distance using square of tunnel-through distance.

Sometimes only latitude and longitude coordinates are available. Converting latitude and longitude to Cartesian coordinates is performed with the formulas:

$$\begin{aligned}
 x &= r \cos(\lambda) \cos(\phi) \\
 y &= r \sin(\lambda) \cos(\phi) \\
 z &= r \sin(\phi).
 \end{aligned}$$

Since the calculation of the Cartesian coordinates for two points is about as expensive as the calculation of the great-circle distance, the Cartesian coordinates should be calculated once for each point and stored.

## 4 Optimizing the Great-Circle Distance Calculation

The time spent calculating the great-circle distance occurs almost entirely within the calculation of the central angle,  $\theta$ . Therefore, that is the calculation to optimize. Trigonometric operations are usually considerably more expensive than normal floating point operations, and the haversine formula contains five of them. The author seeks a way to optimize the calculation of the central angle by reducing the number of trigonometric operations. The chord function for a circle is

$$\text{crd } \theta = 2r \sin \frac{\theta}{2}.$$

Consider the right diagram of Figure 1, which shows a slice of a great circle of the earth. Noting that  $d$  is the chord yields a chord function of

$$d = 2r \sin \frac{\theta}{2},$$

and solving for  $\theta$  gives us

$$\theta = 2 \arcsin \frac{d}{2r}.$$

This yields the following formula for great-circle distance:

$$s = 2r \arcsin \frac{d}{2r}.$$

```

double greatCircleDistance(const Point& p1, const Point& p2)
{
    const double radius2 = 7912; // 2 * average radius of earth
    const double invRadius2 = 1.0 / radius2;

    double a = sqrt(comparableDistance(p1, p2)) * invRadius2;
    if (a > 1) a = 1;

    return radius2 * asin(a);
}

```

**Listing 3.** Optimized great-circle distance calculation.

To protect against undefined arcsine values, the parameter to arcsine should again be wrapped in a min function yielding the following formula:

$$s = 2r \arcsin \left( \min \left[ 1, \frac{d}{2r} \right] \right).$$

Since it has already been shown how to calculate  $d$  inexpensively when the Cartesian coordinates of the points are available, this formulation is less expensive than the haversine formula. Both formulas require one square root and ten floating point operations. However, the optimized formula has only one trigonometric operation compared to the haversine formula's five. Thus, the optimized formula has a savings of four trigonometric operations. Listing 3 shows C++ code that performs the computation efficiently.

## 5 Optimizing the Nearest Neighbor Search Using Spatial Data Structures

The simple way to perform a nearest neighbor search is a linear search, comparing point  $q$  with every point in the set  $P$ . However, methods that compare less points are possible if the points in  $S$  are related spatially. Many data structures that take advantage of spatial information exist and can be used to reduce the number of points tested in  $P$ . Examples of spatial data structures include quadtrees, octrees, k-d trees, r-trees, and grid partitioners. The author considered and tested several versions of grid partitioners and k-d trees.

The general idea of a spatial data structure is to partition the space  $S$ , and consequently the points in  $P$ , such that a small subset of  $P$  needs to be considered during a search. A search starts in the partition that contains  $q$  and expands to neighboring partitions as necessary. The search can stop when a bounds test proves that the nearest neighbor is contained in the space represented by the partitions searched so far. Operations performed by a spatial data structure include finding the partition that contains a given point in  $S$ , finding the neighbors of a partition, and the bounds test.

The search cost is roughly equal to the sum of the cost of determining  $q$ 's partition, the cost of

determining neighboring partitions, and the cost of searching the points in the examined partitions. The spatial data structures described in this paper utilize a maximum number of points per partition to determine the number of partitions. Let  $g$  be the maximum number of points per partition. The best running time occurs when the points are spread evenly across all the partitions. In this case the expected number of points examined is  $O(g)$ , regardless of the size of  $P$ . Many searches will terminate after searching one partition and few, if any, should search more than a small constant number of partitions. Thus, the best case time for a partitioner is independent of the problem size. However, the number of points examined by a search has a worst case of  $O(|P|)$  that occurs when the points are concentrated in a few partitions.

Since the running time in the best case depends on only  $g$ , one might think that setting  $g$  to 1 would achieve the best performance. Even though having a few number of points per partition usually causes fewer points to be examined, the number of partitions examined in a search increases. As there are costs associated with both examining points and examining partitions, reducing  $g$  too much can cause a search to run slower.

Another issue that affects running time is the location of  $q$  in comparison to the points in  $P$ . Let  $T$  be a bounded version of  $S$  that is created by placing a bounding box around all the points in  $P$ . When  $q$  is located far away from any of the points in  $P$ , whether  $q$  is inside or outside of  $T$ , the running time can be adversely affected because a greater number of partitions may be searched.

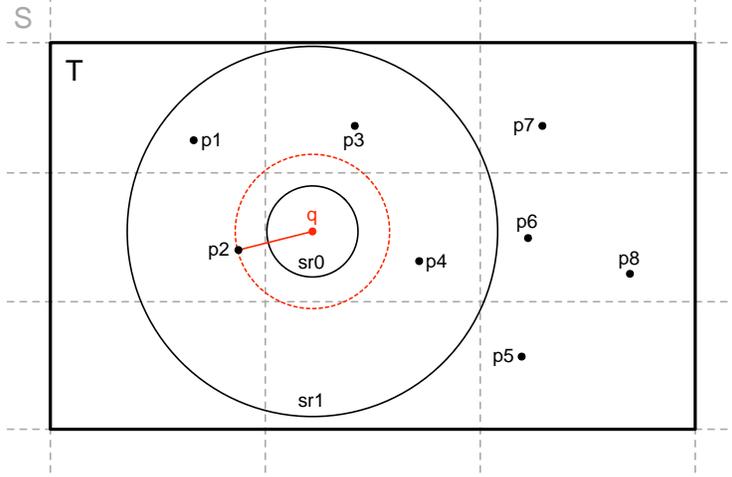
## 5.1 Grid Partitioners

Let  $k$  be the number of dimensions in the space  $S$ . Partition each dimension of  $S$  into even divisions creating identically sized  $k$ -dimensional rectangular block partitions. The size of the partitions is chosen to ensure that the average points per partition is at most  $g$ . Each point is located in a single partition. Points that occur exactly on a division are consistently placed in the “left” partition.

Let the partition in which  $q$  lies be called level 0. Level  $n$  is all of the surrounding partitions that touch level  $n - 1$  even if only at a corner. The levels are nested shells that have a thickness of one partition. The search occurs in phases where phase  $n$  is associated with level  $n$ . The search radius for a phase is the distance from  $q$  to the nearest outer edge of its level. Phase 0 examines the points in level 0 that lie within its search radius. Phase  $n$  examines the points that lie between the search radius of phase  $n - 1$  and the search radius of phase  $n$ . Note that the points could lie in partitions from level  $n$  or level  $n - 1$ . If a point lies exactly on a phase’s search radius, it is considered in the next phase. If no points from  $S$  lie within a phase’s search radius, the search continues to the next phase. A phase must examine all of its points to guarantee that the solution point is the nearest.

Consider the 2D grid partitioner shown in Figure 2 where  $S = \{p1, \dots, p8\}$ . Dotted gray lines indicate partition divisions. Level 0 is the center partition which contains  $q$ . Level 1 is the eight partitions surrounding the center partition. The search radius for phase 0 is shown as  $sr0$ , while the search radius for phase 1 is shown as  $sr1$ . In phase 0 of the search, there are no points within  $sr0$ , so the search continues to phase 1. In phase 1 there are four points to choose from that lie between  $sr0$  and  $sr1$ :  $p1$ ,  $p2$ ,  $p3$ , and  $p4$ . The search examines all four points and finds that  $p2$  is the closest point.

The author tested both 2D and 3D grid partitioners for N-ABLE<sup>TM</sup>. Even though points on



**Figure 2.** 2D grid partitioner example.

the earth really exist in 3D space, they can be considered to exist in a 2D space that is wrapped around a sphere. The coordinate system in this 2D space is latitude and longitude. Points are assigned to partitions using latitude and longitude; however, latitude and longitude do not provide a good distance metric. The great-circle distance is the appropriate metric, but the tunnel-through distance can be substituted as a less expensive magnitude-comparable distance.

Unfortunately, the 2D grid partitioner has problems. Using latitude and longitude for partitioning introduces some error as longitudinal lines are not parallel. Near the equator the error is minimal, but it becomes quite significant nearing the poles. Additionally, the partitioner must wrap the space in both dimensions. Wrapping in the longitudinal direction happens neatly, but wrapping in the latitudinal direction does not make much sense. The latitudinal “line” converges to a point at the poles causing all the partitions surrounding a pole to touch. The meaning of neighboring partitions is a bit vague in this circumstance. Additionally, using great-circle distance to calculate the search radius creates a circle that is slightly skewed with respect to the latitude and longitude search space. However, a reasonably accurate nearest neighbor search can still be performed with a 2D grid partitioner if neither  $q$  nor any of the points in  $P$  are close to the poles.

A 3D grid partitioner solves the error problems of a 2D grid partitioner as it partitions points based on Euclidean space and uses a Euclidean metric for distance. Unfortunately, it can be considerably slower than the 2D grid partitioner. In 3D the number of partitions in a level grows much faster than in 2D. Let  $l$  be the current level where level 0, the first level, has 1 partition. The level  $l$  is a border that defines an  $n$ -cube with  $(2l + 1)^n$  partitions. The number of partitions in level  $l$ , where  $l > 0$ , is equal to the difference between the number of partitions in level  $l$ 's  $n$ -cube and level  $(l - 1)$ 's  $n$ -cube. For 2D and 3D grid partitioners, the number of partitions in level  $l$  is

$$\begin{aligned}
 2D : & \quad (2l + 1)^2 - (2l - 1)^2 = 8l \\
 3D : & \quad (2l + 1)^3 - (2l - 1)^3 = 24l^2 + 2,
 \end{aligned}$$

which means the number of partitions grows linearly with increasing levels in 2D but quadratically in 3D. If  $q$  is chosen such that it lies in a partition with points, then the 2D and 3D grid partitioners

will perform the same, but the 3D grid partitioner will perform much worse if  $q$  is chosen such that it lies many partitions away from a partition that contains points.

Thus, one important issue affecting the running time of a grid partitioner is the location of  $q$ . When  $q$  is located in a partition that is many partitions away from the nearest point, many levels of empty partitions must be examined before a partition containing points is found. Additionally, the number of points initially considered may be large as a high level contains many partitions. This occurs when  $q$  is outside of  $T$ . It can also occur when  $q$  is inside  $T$  and the points are concentrated in a few small areas.

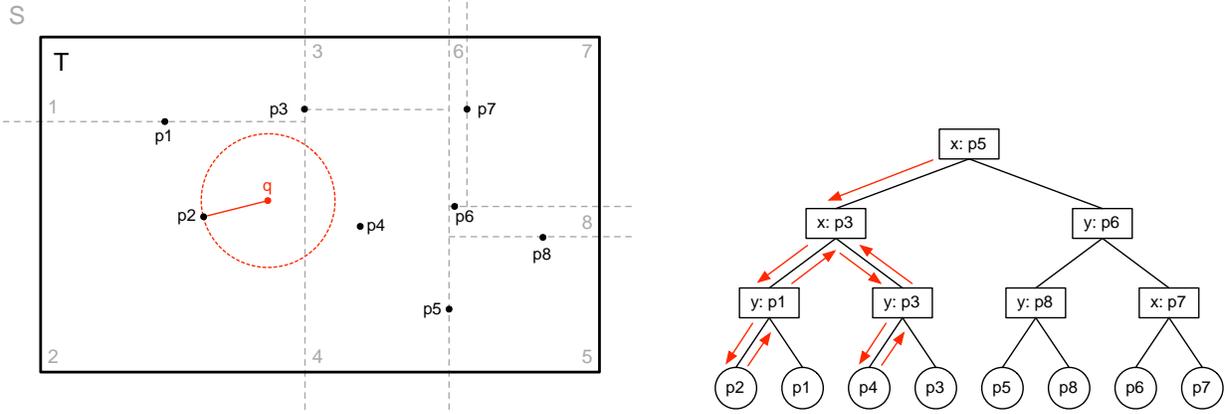
Another important issue affecting the running time of a grid partitioner is the distribution of the points across the partitions in  $T$ , which affects the number of points examined during a search. The fewest number of points the grid partitioner can examine is  $O(g)$ . This occurs when  $q$  is inside  $T$  and the points are spread evenly across all the partitions. Unfortunately, the grid partitioner can suffer a worst case of  $O(|P|)$  points examined during a search when the points are concentrated in a few small areas that are far apart. One situation that would cause this worst case occurs when a 2D partitioner has a single point located in the top left partition of  $T$  and all the remaining points are located in the bottom right partition of  $T$ . A  $q$  in or near the bottom right partition could require a search through the  $|P| - 1$  points in that partition.

## 5.2 K-d Trees

A k-d tree [1, 4] is a type of binary tree that provides fast geospatial queries for multidimensional point data. Each node in the tree represents a partition of the space  $S$  that defines a subset of the points  $P$ . The root node represents the entire space  $S$ , and each child node represents a partition of its parent's space. Each leaf node stores all the points defined by its partition of  $S$ . A non-leaf node stores no points.

A k-d tree is created as follows. Set a maximum points per partition that defines the maximum number of points that can be stored in a leaf node. Starting with the root node, choose the dimension of the node's space that has the largest range. This dimension is called the partition dimension. Find the point in the subset of  $P$  defined by the node's space that has the median value for the partition dimension. Partition the node's space in the partition dimension at the median value. The node's left child receives the subdivision of the space that is less than the median, and the right child receives the subdivision of the space that is greater than or equal to the median. For the partition dimension, the left child has boundaries of [parent's left boundary, median) while the right child has boundaries of [median, parent's right boundary]. For all other dimensions, the children inherit their parent's boundaries. Now, partition the child nodes' spaces. Continue the partitioning until the number of points defined by a node's space is less than the maximum points per partition. At this point create a leaf node. Each leaf node stores the set of points that fall within the boundaries defined by its space.

To perform the nearest neighbor search, search the tree in a depth-first fashion. The search starts at the root node and proceeds down the tree to the leaf node whose boundaries contain  $q$ . It performs a linear search of all of the leaf's points to find the closest point. Let the search sphere be the  $n$ -sphere with center  $q$  and radius of the distance between  $q$  and the current closest point. If the search sphere is contained entirely within the leaf's boundaries, the search can terminate. Otherwise, a point may exist in another node that is closer, and the search travels back up the



**Figure 3.** K-d tree: division of  $S$  space and the generated tree.

tree. When the search returns to a parent node after checking its first child, it checks if the search sphere overlaps the other child’s boundaries. If so, the other child is searched down to a leaf, and the new leaf’s points are checked for a closer point. If one is found, the search updates the closest point and search sphere accordingly. Once both a parent node’s children have been considered, the search can terminate if the search sphere is contained entirely within the node’s boundaries. Otherwise the search must continue back up the tree.

Consider Figure 3, which is the same example as shown in Figure 2 for the grid partitioner. For this 2D example the maximum points per partition is 1. The left diagram shows how the search space is divided by the tree. Dotted gray lines indicate partition divisions. Gray numbers are the partition numbers. Point  $p_n$  is in partition  $n$ . The right diagram shows the generated tree. Leaf nodes show the point they contain. Non-leaf nodes show the dimension partitioned on and the point whose value in that dimension was the median. The red arrows show the path the search follows when looking for the nearest neighbor to  $q$ .

The search for  $q$  happens as follows. The search starts at the root and follows the left side of the tree down to the leaf that contains  $p_2$ . This is the leaf in whose boundaries  $q$  is located. The points of this leaf are searched, and  $p_2$  is the closest point. Since the search sphere is not contained entirely in the bounds of the node, the search proceeds to the parent. The other child does not need to be checked, but the search sphere is still not contained within the node’s boundaries. The next parent, which is the left second level node, is checked, and its other child needs to be searched. The search proceeds down to the leaf containing  $p_4$ . None of the points in node are closer than  $p_2$ , so  $p_2$  remains the closest point. Since the search sphere is not contained entirely in the bounds of the node, the search proceeds to the parent. Again, the search sphere is not contained entirely in the bounds of the node, so the search proceeds to the next parent which is the left second level node. The search sphere is contained entirely within the node’s boundaries, so the search stops, yielding  $p_2$  as the closest point.

The implementation of the k-d tree described in this paper is the one from [4]. There is a slightly different implementation described in [1] which assigns a single point to each node in the tree including internal nodes. The author tested this version of the k-d tree as well. It had similar

performance, but the first tree performed slightly better.

The author tested a three-dimensional k-d tree for N-ABLE<sup>TM</sup>. The 3D k-d tree has several improvements over the grid partitioners. It does not suffer from any of the accuracy errors that the 2D grid partitioner does, and it is much faster than both of the grid partitioners. One reason is because its partitions are dynamically sized at creation to evenly divide the points between the partitions while the grid partitioner’s partitions are statically sized. A k-d tree contains no empty partitions no matter how the points are distributed across  $S$ . Thus, the number of points examined is expected to be  $O(g)$  as long as  $q$  is inside  $T$ . Unfortunately, many partitions can still be examined when  $q$  is far outside of  $T$ , but that number should be considerably less than the number of partitions considered by a grid partitioner as a k-d tree has no empty partitions.

## 6 Implementation in N-ABLE<sup>TM</sup>

The NISAC Agent-Based Laboratory for Economics (N-ABLE<sup>TM</sup>) is an agent-based discrete-event microeconomic simulation tool that captures complex internal supply chain and market dynamics of businesses in the U.S. economy. Firms are modeled as economic agents that produce and consume commodities and buy and sell them from each other. N-ABLE<sup>TM</sup> models the shipping of commodities as packages sent between buyers and sellers using a shipping agent. The shipping agent provides several types of transportation modeling. The simplest is a diffusion mode that models the movement of packages along a straight line between the buyer and seller. The shipping agent also provides a network mode that models the movement of packages along a transportation network. The network mode captures the flows of traffic along the infrastructure and can simulate disruptions to the infrastructure. The infrastructure network used in N-ABLE<sup>TM</sup> is an intermodal network comprised of truck, rail, and water subnetworks connected by intermodal terminals. This network is a slightly modified version of an intermodal network created by the Center for Transportation Analysis at Oak Ridge National Laboratory [5].<sup>1</sup> The truck and rail networks represent the truck and rail networks in North America. The water network represents inland water routes in North America plus ocean routes across the globe.

The shipping agent performs both the calculation of the distance between two points on the globe and the nearest neighbor search. The agent calculates the distance between two points when sending a package using the diffusion mode. The shipper performs a nearest neighbor search when using the network mode as the shipper must find the nearest transportation vertices to the buyer and seller. These two calculations are performed many times by the shipper, so the author desired to make them run as fast as possible. In fact, the initial implementation of the nearest neighbor search, which utilized the simple linear search and great-circle distance, took a noticeable percentage of the total simulation time. Currently, N-ABLE<sup>TM</sup> uses the optimized distance calculation described in this paper when calculating the distance between two points for the diffusion mode. It uses a 3D k-d tree with a distance metric of the square of the tunnel-through distance to perform nearest neighbor searches for the network mode. After including the optimizations, the nearest neighbor search took a minuscule fraction of the runtime.

Since the vertices in the networks are static throughout a simulation, extra initialization time is acceptable as the initialization is performed once. A moderate amount of extra storage is also

---

<sup>1</sup>The version of the network on the website is cks02. N-ABLE<sup>TM</sup> uses a modified version of the newer ck34, which was mailed directly to the author by Bruce Peterson.

acceptable as a tradeoff for speed improvements. Each agent in N-ABLE<sup>TM</sup> has a location, which stores geographic coordinates (latitude and longitude). The transportation network vertices also have locations. The enhancements described in this paper require the calculating and storing of 3D Cartesian coordinates in addition to the geographic coordinates. The k-d tree requires additional storage and the time cost of initializing the tree. The time cost of initializing the Cartesian coordinates and the k-d tree are small compared to the time savings during a normal simulation run, and the space cost is very modest compared to the memory already consumed by the simulation. The maximum points per partition was set to 30 for the k-d tree as this seemed to provide the best overall performance when considering  $q$ 's that were both inside and outside of  $T$ .

## 7 Testing and Results

The tests were performed on an Apple Mac Pro with a single 2.8 GHz Intel Xeon quad-core processor running OS X 10.5 and using GCC v4.0.1, the version of the GCC compiler that comes with Xcode 3.1. The test codes were compiled with O2 optimizations. The timings do not include the initialization time for computing the Cartesian coordinates or the time for initializing the data structures. The maximum points per partition for all the spatial data structures was set to 30 for the tests, and the spatial data structures use the magnitude-comparable distance as their distance metric.

The tests use two sets of locations for the set of points  $P$ : the vertices from the N-ABLE<sup>TM</sup> rail network and the vertices from the N-ABLE<sup>TM</sup> truck network. The rail network has 14,680 vertices and the truck network has 67,790 vertices. For  $q$  the tests use two sets of points. The first set is the land points set, which is a grid of points bounded by three rectangles placed to attempt to cover most of the land in the US. The three rectangles are defined by the following sets of latitude and longitude coordinates:

$$\begin{aligned} & (46.5, -118.3), (46.5, -88.0), (33.8, -118.3), (33.8, -88.0) \\ & (33.8, -113.0), (33.8, -88.0), (31.5, -113.0), (31.5, -88.0) \\ & (41.5, -88.1), (41.5, -79.0), (30.0, -88.1), (30.0, -79.0). \end{aligned}$$

A point is placed every .1 degrees in both directions, creating 54,961 points. The land points represent well-behaved queries and the typical range of queries an N-ABLE<sup>TM</sup> simulation will provide for  $q$ . The second set of points used for  $q$  is the water points set, which is a similarly sized grid of points bounded by three rectangles placed to represent locations off the US west coast, off the US east coast, and in the Gulf of Mexico. The three rectangles are defined by the following sets of latitude and longitude coordinates:

$$\begin{aligned} & (48.0, -152.0), (48.0, -129.0), (25.0, -152.0), (25.0, -129.0) \\ & (26.0, -93.0), (26.0, -86.0), (24.0, -93.0), (24.0, -86.0) \\ & (38.7, -70.0), (38.7, -69.3), (35.0, -70.0), (35.0, -69.3). \end{aligned}$$

A point is placed every .1 degrees in both directions creating 54,846 points. The water points represent a range of queries that cause the geographical data structures to perform poorly because  $q$  is outside  $T$ .

Table 1 shows the results for the different distance functions. The numbers in the left column for each distance metric are the times in seconds for each test to complete. The numbers in the

Network	Test Points	Haversine GCD		Optimized GCD		MCD	
Rail	Land	120.991	14,680	54.375	14,680	6.794	14,680
Rail	Water	133.326	14,680	64.210	14,680	6.781	14,680
Truck	Land	558.927	69,790	254.759	69,790	34.226	69,790
Truck	Water	613.198	69,790	299.369	69,790	36.155	69,790

**Table 1.** Great-circle and magnitude-comparable distance running times (secs) and points examined.

Network	Test Points	2D Grid		3D Grid		3D K-d tree	
		Partitioner		Partitioner			
Rail	Land	0.312	45.2	0.397	66.4	0.029	53.4
Rail	Water	1.468	37.3	6.214	45.7	0.354	836.3
Truck	Land	0.266	29.4	0.363	53.7	0.032	36.3
Truck	Water	9.521	42.4	94.283	88.1	2.623	5264.5

**Table 2.** Nearest neighbor search running times (secs) and points examined using different spatial data structures.

right column are the average number of points examined by each test. If the exact great-circle distance is needed, then the optimized GCD calculation can be used. In all the tests, the optimized GCD calculation performed more than twice as fast as the haversine GCD calculation. If the magnitude-comparable distance is sufficient, then greater speeds can be achieved. In all the tests, the magnitude-comparable distance calculation performed about 20 times faster than the haversine GCD calculation. As all of these searches are linear, the number of points examined during each search is equal to the number of points in  $P$ . The times for the water points are only slightly higher than the times for the land points. As expected, the linear searches perform roughly the same regardless of where  $q$  is located.

Table 2 shows the results for performing the nearest neighbor search using the different spatial data structures. The numbers in the left column for each distance metric are the times in seconds for each test to complete. The numbers in the right column are the average number of points examined by each test. The spatial data structures perform much better than the MCD linear search for the land points, and each structure has roughly the same performance for the land points regardless of the number of points in  $P$ . The 2D grid partitioner and the k-d tree perform better than the linear search for the water points, but the 3D grid partitioner performs as bad or worse. This fits with the analysis in Section 5.1 of the growth of the search space for grid partitioners when  $q$  is not near partitions that contain points.

The partitioners search far fewer points on average for a search than the linear search. The grid partitioners consistently search a small number of points. The k-d tree searches a similar number of points during the land tests as the grid partitioners, but it searches a more sizable number of points during the water point tests. However, it still takes less time than the grid partitioners.

The k-d tree performs much better than the linear and grid partitioner searches. Its search times are the fastest in every category. It doesn't suffer from the accuracy errors that can occur in

the 2D grid partitioner, and it doesn't suffer from the same horrible performance for difficult data as the 3D grid partitioner. For the land points, the k-d tree performed over 200 times faster and over 1,000 times faster than the linear search for the rail and truck networks, respectively. For the water points, it performed an order of magnitude faster than the linear search.

N-ABLE<sup>TM</sup> originally used a linear search with the haversine GCD metric to perform nearest neighbor searches. The combination of switching to the MCD and the k-d tree yielded an enormous speedup to the searches. For the land test points, we achieved speedups of three and four orders of magnitude for the rail and truck networks, respectively. For the water test points, we achieved speedups of two orders of magnitude.

## 8 Summary

The author has described how to use a magnitude-comparable distance to speed up the nearest neighbor search and the calculation of great-circle distance. The author has also described spatial data structures that can further speed up the nearest neighbor search. The improvement to the distance metric requires Cartesian coordinates for the points which can be precalculated and stored if they are not already available. The k-d tree requires additional storage and the one-time cost of initializing the tree.

The new method for calculating great-circle distance performed twice as fast as the standard method. Using a 3D k-d tree with the magnitude-comparable distance for the nearest neighbor search performed four orders of magnitude better in the best case and two orders of magnitude better in the worst case compared to the original method using a linear search with the great-circle distance. Including these optimizations in N-ABLE<sup>TM</sup> changed the time spent performing nearest neighbor searches from a noticeable percentage of the simulation runtime to a minuscule fraction of the runtime.

## 9 Acknowledgments

This work was performed with funding from the DHS Science and Technology Directorate.

## References

- [1] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.
- [2] R. G. Chamberlain. Geographic Information Systems FAQ 5.1. Originally posted to comp.infosystems.gis newsgroup, April 1997. <http://www.faqs.org/faqs/geography/infosystems-faq/> accessed on January 27, 2010.
- [3] E. D. Eidson and M. A. Ehlen. Nisac agent-based laboratory for economics (N-ABLE<sup>TM</sup>): Overview of agent and simulation architectures. SAND Report, 2005.
- [4] J. H. Freidman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3):209–226, 1977.
- [5] B. E. Peterson. Intermodal Transportation Network. Oak Ridge National Laboratory - Center for Transportation Analysis, August 2000. [http://cta.ornl.gov/transnet/Intermodal\\_Network.html](http://cta.ornl.gov/transnet/Intermodal_Network.html) accessed on January 27, 2010.
- [6] R. W. Sinott. Virtues of the haversine. *Sky and Telescope*, 68(2):159, 1984.

## DISTRIBUTION:

- 1 MS 1138 Eric Eidson, 06371 (electronic copy)
- 1 MS 1138 Mark Ehlen, 06371 (electronic copy)
- 1 MS 1137 Greg Mackey, 06371 (electronic copy)
- 1 MS 1138 Lillian Snyder, 06371 (electronic copy)
- 1 MS 1138 Eric Vugrin, 06371 (electronic copy)
- 1 MS 0899 Technical Library, 9536 (electronic copy)







**Sandia National Laboratories**