Multilateration Index - Masters Thesis

Chip Lynch

03/24/2021

Contents

A Multilateration Alternate Coordinate System

Abstract

We present an alternative method for pre-processing and storing multilateration point data, particularly for Geospatial points, by storing distances to fixed points rather than coordinates such as Latitude and Longitude.

We explore the use of this data to improve query performance for some distance related queries such as nearest neighbor and query-within-radius (i.e. “find all points in a set P within distance d of query point q”).

Our construction includes storing the distances from fixed points (typically three, as in trilateration) as an alternative to Latitude and Longitude. This effectively creates a coordinate system where the coordinates are the trilateration distances. We explore this alternative coordinate system and the theoretical, technical, and practical implications of using it. Multilateration itself is a common technique in surveying and geolocation widely used in cartography, surveying, and orienteering, although algorithmic use of these concepts for NN-style problems are scarce. GPS uses the concept of detecting the distance of a device to multiple satellites to determine the location of the device; a concept known as true-range multilateration. However while the approach is common, the distance values from multilateration are typically immediately converted to Latitude/Longitude and then discarded. Here we attempt to use those intermediate distance values to computational benefit. Conceptually, our multilateration construction is applicable to metric spaces in any number of dimensions.

Rather than requiring the complex pre-calculated tree structures, or high cost pre-calculated nearest-neighbor sub-trees (as in FAISS), we rely only on sorted arrays as indexes. This approach also allows for processing computationally intensive distance queries (such as nearest-neighbor) in a way that is easily implemented with data manipulation languages such as SQL, where we see a roughly 10x performance improvement to existing query logic. Other modern nearest-neighbor solutions such as KD-Tree, Ball-Tree, and FAISS require capability beyond that of standard SQL to be performant (such as efficient implementations of tree-structures).

Outside of SQL, our approach shows significant performance gains - 30x performance using the ann-benchmark tool for nearest-neighbors - when the cost of the atomic distance calculation itself is high, such as with geodesic distances on earth using accurate elliptical, rather than spherical or euclidean models of the earth. Of note, our algorithms do not improve on the time-complexity of nearest-neighbor searches; we remain $O(n)$ in the worst case. The main point of improvement is that we can substitute simple subtraction for the distance function itself, after performing $d * n$ distance calculations to create our initial multilateration structure. In effect, remembering that $O(C * f(x)) == O(f(x))$ where C is a constant, our approach focuses on reducing C rather than reducing the time complexity of $f(x)$ itself.

Further, we discuss the problem of “Network Adequacy” common to medical and communications businesses, to analyze questions such as “are at least 90% of patients living within 50 miles of a covered emergency room”. This is in fact the class of question that led to the creation of our pre-processing and algorithms, and is a generalization of a class of Nearest-Neighbor problems.

While we focus primarily on geospatial data, potential applications to this approach extend to any distance-measured n-dimensional metric space, and we touch on those (briefly, here, to constrain our scope). For example, we consider applying the technique to Levenshtein distance, MINST datasets via cosine-similarity, and even facial recognition or taxicab systems where distances do not follow the triangle inequality.

1 Review of Current Literature

We divide our current literature review into sevveral major sections:

1. Geospatial computational considerations
2. Multilateration
3. Nearest Neighbor Algorithms
4. Ancillary references outside those two areas:

1.1 Geospatial Computations

The key to our problem is accurate distance calculation on the Earth. In particular, we explore the topic of the computational complexity of a single call to the distance function for geospatial distances.

In general, the difficulty of determining distances on the earth goes back to the origins of sailing and before, with modern geography tracing its origins to Eratosthenes, whose claim is to be the first to calculate the circumference of the earth arund 200BC, and who himself was building on the ideas of a spherical earth from Pythagoras and others around 500BC. (Kaplan 2018)

1.1.1 Haversine

The Haversine is the measurement of distances along a sphere; particularly great-circle (minimum) distances on a spherical earth's surface. Tables of the Haversine were published around 1800AD, although the term was not coined until aroudn 1835 by James Inman.(Brummelen 2012)

Given the radius of a spherical representation of the earth as $r = 6356.752km$ and the coordinates of two points (latitude, longitude) given by (ϕ_1, λ_1) and (ϕ_2, λ_2), the distance d between those points along the surface of the earth is (Gade 2010):

$$d = 2r\sin^{-1}(\sqrt{\sin^2(\frac{\phi_2 - \phi_1}{2}) + \cos(\phi_1)\cos(\phi_2)\sin^2(\frac{\lambda_2 - \lambda_1}{2})})$$

Obviously this is somewhat computationally complex, comprising five trigonometric functions, two subtractions and a square root. While it is a closed form solution, it causes an error over long distances of up to 0.3%, which can mean distances are off by up to 3 meters over distances of 1000 kilometers. From the equator to the north pole, which on a sphere is defined as precisely 10,000 km, the actual distance is off by over 2 km, which is a sizeable error for even the most robust applications.

1.1.2 Vincenty's Formula

The shortcomings of the spherical calculation was thoroughly discussed by Walter Lambert in 1942.(Lambert 1942) However it wasn't until 1975 that an iterative computational approach came about to give more accurate distance measurements with a model of the earth more consistent with reality. By considering the earth as an ellipsoid, rather than a sphere, the distance calculations are more complex, but far more precise. Vincenty was able to create an iterative approach accurate down to the millimeter level on an ideal elliptical earth; far more accurate than the Haversine calculations(Vincenty 1975). This algorithm, however, was a series which failed to converge for points at near opposite sides of the earth.(Karney 2013)

1.1.3 Karney's Formula

Karney was able to improve upon this in 2013 to fix these antipodal non-convergences, and the resulting formulae are now widely available in geospatial software libraries where precision is required (commonly referred to as "Geodesic" distances. (Karney 2013) This is currently the state-of-the art implementation of precise geospatial distances. Implementations of this approach have already been implemented in Python (in the Geopy library), which we use in our Python implementations.(Brian Beck, n.d.)

1.2 Multilateration

There are numerous uses of multilateration as an approach to determining the position or location of real world objects including:

- GPS (Abel and Chaffee 1991)
- RFID (Zhou and Shi 2009), (Zhang et al. 2017) ** And wireless networks in general (Singh and Sharma 2015)
- Machine Tooling and Calibration (Linares et al. 2020)
- Air Traffic Control (Strohmeier, Martinovic, and Lenders 2018), (Kazel 1972)
- Machine Learning (Tillquist and Lladser 2016)

Many of these papers describe mechanisms for taking multilateration measurements, from Radar, Wireless Networks, Lasers, Satellites, etc. and transforming them into another coordinate system, and acting on that information. The cost of this transformation is itself expensive, requiring iterative numerical techniques for real world solutions.(Lee 1975)

1.3 Nearest Neighbor

The nearest neighbor (NN) problem should need no introduction. For our perspective, we talk about NN and k-NN as:

Given a non-empty set of points P, a non-empty set of query points Q in a metric space M, and a distance function $D(a, b)$ describing the distance between points a and b for $a \in M$ and $b \in M$, the "Nearest Neighbor" of a given point $q \in Q$ is the point $p \in P$ such that $D(p, q)$ is the lowest value of $D(p', q)$ over all points $p' \in P$ (i.e. $D(p, q) < D(p', q) \forall p'$ with $p \neq p'$.

Note that it is possible that such a point does not exist if there are multiple points with the same lowest distance; we do not explore that situation here, as it does not affect our examination.

The k-nearest neighbors (kNN) of a given point q as above is the list of k points $R = p_1..p_k \in P$ such that $D(p_k, q)$ is the lowest value of $D(p, q)$ over all $p \in P$ such that $D(p', q)$ for $p' \neq p$ and $p' \notin R$. It should be evident that $NN = kNN$ when $k = 1$.

An approximate nearest neighbor (ANN) algorithm is one which will provide k points $R' = p_1..p_k \in P$, however it does not guarantee that there exists no point in $P \notin R'$ closer than any point in R'. A $c - ANN$ formulation, for example, requires that, if there is such a point p', that it cannot be more than some $\epsilon < c$ farther from q than any point $p_i \in R'$; that is, a solution must guarantee that $d(p, q) < c * d(p', q)$ In general, ANNs are used when we can get a 'close enough' solution algorithmically faster than a perfect kNN solution. For our purposes, we largely ignore ANNs except for their historical value, as our construction did not yield algorithms that exhibited this beneficial tradeoff. See the "TrilatApprox" algorithm section for some more discussion.

For clarity in this paper we use the common notation $|P|$ and $|Q|$ to refer to the number of points in P and Q respectively.

1.3.1 Comparing Algorithms

Solutions for NN queries can be compared across a variety of metrics which, when possible, we explore for each algorithm:

1. Training Time Complexity - the $O()$ required to pre-process the points p if any

2. Memory Space - the memory requirements (typically in terms of $|P|$) of the structures resulting from pre-processing
3. Prediction Time Complexity - the $O()$ required to find the $kNN \in P$ for a single point q
4. Insertion/Move Complexity - the $O()$ complexity required to add or move (or remove) a point $p \in P$

In some cases these are directly calculable theoretically, however many algorithms suffer from theoretical worst-case situations that are not realistic. Synthetic benchmarks such as “ANN-Benchmark” (which we use to report our experimental results) exist for this reason.(Aumüller, Bernhardsson, and Faithfull 2020)

In general we want some standard bounds on these values. Our list here is compatible with (Chen and Shah 2018), which sets the following bounds:

> ideally we would like nearest neighbor data structures with the following properties:
>
> 1. Fast [Prediction Time Complexity]. The cost of finding k nearest neighbors (for constant k) should be sublinear in n, i.e., o(n); the smaller the better.
> 2. Low storage overhead [Memory Space]. The storage required for the data structure should be subquadratic in n, i.e., $o(n^2); the smaller the better.
> 3. Low pre-processing [Training Time Complexity]. The cost of pre-processing data to build the data structure should not require computing all pairwise distances and should thus be o(n^2); the smaller the better.
> 4. Incremental insertions [Insert Complexity] It should be possible to add data incrementally to the data structure with insertion running time o(n).
> 5. Generic distances and spaces. The data structure should be able to handle all forms of distances ρ and all forms of spaces X.
> 6. Incremental deletions [Move Complexity] The data structure should allow removal of data points from it with deletion running time o(n).

We don’t address point 5 from (Chen and Shah 2018); we compare Euclidean, Angular, and Geodesic distances with the ANN-Benchmark software; other distance functions should be generally compatible with all approaches here, but we do not attempt to address all possible distances.

Also, while worst-case computational analysis of NN algorithms is necessarily pessimistic, theoretical differences for average performance are impacted by the relationship between the dimension d and the sample size n… three situations are identifiable, when a point set is: “dense with $d << log(n)$; sparse with $d >> log(n)$; moderate with $d = \Theta(log(n))$”.(Prokhorenkova 2019)

1.3.2 A History of k-NN Solving Algorithms

1.3.2.1 Brute-Force

The naive approach to solving k-nn is a brute-force algorithm, iterating over every point $pinP$ and keeping track of the lowest k distances. This is trivial to examine:

1. Training Time Complexity: Zero; i.e. $O(1)$ No pre-processing is performed.
2. Memory Space: $|P|$, which is simply the cost of storing the list of points $p \in P$
3. Prediction Time Complexity: $O(n)$ - each query must process every point; this is not a worst case, but the every-time case for brute force
4. Insertion/Move Complexity: $O(1)$ - a list element can be added to the end of an array, or a location can be updated in place. There is no complexity to changing a point.

1.3.2.2 Space Partitioning Trees

Space paritioning trees use a trie to arrange points from p into groups with a hierarchical search structure, such that, generally, points which are close to one another exist in nearby hierarchies. The $k-d$ tree was described in 1975.(Bentley 1975) This partitions a space by dividing the underlying points at the median point along one of the dimensional axes, recursively, resulting in a searchable trie. An adaptation of this - the

Ball-Tree - partitions the space into hyperspheres, rather than along dimensional axes.(Liu, Moore, and Gray 2006)

These are straightforward structures that are easy to describe and implement.

Per (Chen and Shah 2018) space partitioning trees, such as k-d and ball trees have:

1. Training Time Complexity: $O(|P| * log(|P|))$ (To calculate a binary search tree [BST] along d dimensions)
2. Memory Space: $O(|P|)$ - the BST is space efficient; every point needs to be stored only once
3. Prediction Time Complexity: $O(d * 2^{O(d)} + log(|P|))$ to query the binary search tree; with low d ($d << log(|P|)$), this is efficient, but since the d appears in the exponent, large numbers of dimensions cause significant problems here.

For point 4 (Insertion/Move Complexity), k-d and ball-trees generally provide no approach which preserves the integrity of the first three complexity measurements. While a $O(log(n))$ insertion is possible (inserting in a BST is not atomically difficult, per se), and while a single insert may not really erode the utility of the tree, if new data is repeatedly added which does not match the distribution of the original space partitioning, the tree will become imbalanced and the logarithmic effect previously guaranteed by the original space division which preserves a balanced tree, will fade, leaving the eventual time complexity back to $O(|P|)$, which is typically $>> d$, and therefore worse.(Chen and Shah 2018)

1.3.2.3 Locality Sensitive Hashing

Locality (sometimes "Locally") Sensitive Hashing (LSH) relies on creating a hash function that hashes points into bins with a property that two points with the same hash have a high likelihood of being nearer to each other than points with different hash values.(Indyk and Motwani 1998) Formally:

For a given threshold $\epsilon > 0$, and a hash function $H(p) => S$ maps to a space S for all $p \in P$ * if $d(p, q) < \epsilon$ then $H(p) = Hq$ with probability at least λ_1 * if $d(p, q) > \epsilon$ then $H(p) \neq H(q)$ with probability at MOST λ_2

And a value k which is the approximate number of points $p \in P$ which hash to a given value. (Paulevé, Jégou, and Amsaleg 2010) think of this value as the fraction $sel = k/|P|$: "the selectivity sel is the fraction of the data collection that is returned in the short-list, on average, by the algorithm".(Paulevé, Jégou, and Amsaleg 2010)

LSH as a concept lends itself to no specific complexity analyis, since it is dependent on the particular hash functions chosen. LSH can leverage L multiple hashes, and the selection of the number and type of hashing algorithm is itself the basis of research and variety.(Paulevé, Jégou, and Amsaleg 2010),(Chen and Shah 2018)

1. Training Time Complexity: if the hash function(s) take time $O(t)$, the LSH prep takes $O(|P| * L * O(t))$ - the cost of executing the functions to hash into L buckets against all points $p \in P$
2. Memory Space: a single hash function typically requires $O(|P|)$ space or for L hashes memory space: $O(L * |P|)$.
3. Prediction Time Complexity: This feature is the most highly dependent on the selection of hash function, and the width of the hash buckets (k). The worst case is $O(|P|)$, however for large enough sets given λ_2, the average case can be estimated as: $O(L * O(t) + L * |P| * d * \lambda_2^k)$ - (Prokhorenkova 2019) simplifies this to $O(d * |P|^{\phi})$ for $c - ANN$ algorithms where $\phi \approx \frac{1}{c}$
4. Insertion/Move Complexity: $O(L * O(t))$ - the cost of executing the L hash functions on the new point's data

1.3.2.4 Graph Based Search

More recent algorithms, such as Facebook Research's FAISS, follow a graph based search structure.(Johnson, Douze, and Jégou 2017)

A good overview of this approach was available from Liudmila Prokhorenkova: "Recently, graph-based approaches were shown to demonstrate superior performance over other types of algorithms in many large-scale applications of NNS (Aumüller, Bernhardsson, and Faithfull 2020). Most graph-based methods are based on constructing a k-nearest neighbor graph (or its approximation), where nodes correspond to the elements of D, and each node is connected to its nearest neighbors by directed edges.(Dong, Charikar, and Li 2011) Then, for a given query q, one first takes an element in D (either random or fixed predefined) and makes greedy steps towards q on the graph: at each step, all neighbors of a current node are evaluated, and the one closest to q is chosen."(Prokhorenkova 2019)

The construction costs of these structures can be very high. A Brute Force construction of a k-Nearest Neighbor Graph ($kNNG$) has time complexity $O(n^2)$ which is of course completely untenable for large data sets. Approaches exist to improve upon this, including improvements resulting in approximate results, but this class still tends to trade the highest construction cost for some of the fastest query times in high dimensions.(Dong, Charikar, and Li 2011), (Prokhorenkova 2019)

One interesting point about FAISS is that it is designed to be highly parallelized, and particularly focused on optimizations available to GPUs.(Johnson, Douze, and Jégou 2017) This may be a disadvantage for comparisons with the ANN Benchmark suite that we use, which forces single-threaded operation, in order to test the algorithm rather than the hardware.(Aumüller, Bernhardsson, and Faithfull 2020)

As these are relatively new, and as implementations focus on real-world performance, rather than theoretical, "Graph-based approaches are empirically shown to be very successful for the nearest neighbor search (NNS). However, there has been very little re-search on their theoretical guarantees."(Prokhorenkova 2019)

As with LSH, Graph search methods differ in their specific choices; typically the mechanism for generating the search graph. In general, these graphs are called "Monotonic Search Networks" (MSNETs)… (Fu et al. 2018) compares "Monotonic Relative Neighborhood Graphs" (MRNGs) and "Navigating Spreading-out Graphs" (NSGs).

1. Training Time Complexity: (Fu et al. 2018) reports $O(n^2 \log(n) + n^2 * c)$ for an "MRNG" where c is the average out-degree of [the graph]". For NSGs, this can be reduced to $O(kn\frac{1+d}{d}log(n^{\frac{1}{d}}) + n^{1.16})$ and $O(kn\frac{1+d}{d} \log(n^{\frac{1}{d}}) + n * \log(n))$ for FAISS. Note that these are all by far the largest preparation complexities of all our reviewed algorithms.
2. Memory Space: Efficient graph storage requires storing k edges for each of n points for a base space complexity of $O(kn)$
3. Prediction Time Complexity: (Fu et al. 2018) reports theoretical results for NSGs as $O(cn^{\frac{1+\epsilon}{d}} \log n^{\frac{1}{d}})$, which, for large d approaches $O(\log n)$, which matches their experimental results.
4. Insertion/Move Complexity: This doesn't seem closely studied, but in principle inserting a new row costs the time to re-evaluate the kNN graph for that point, and update any points for which the kNN graph would change with its addition. We expect this should roughly be between $O(k \log n)$ and $O(k^2 \log n)$

1.4 Network Adequacy

We can find no literature where this topic is solved in a particular algorighmic way. There are numerous discussions in health care about satisfying network adequacy, but more as policy or health care topics than as computational approaches. (Wishner and Marks 2017),(Mahdavi and Mahdavi 2011)

In general, it appears that most practical solutions are done in SQL databases which are commonly the source of member and provider data for health care datasets. Still, there is little published here; this information is anecdotal based on the author's personal direct knowledge and informal research.

Satellite and cellular network discussions of this problem appear to be proprietary, but again anecdotally, appear to simply apply common Nearest-Neighbor algorithms.

Where we can find references to actual applications, the implemented solutions tend to be iterative, exhaustive implementations of existing Nearest-Neighbor algorithms.

It is worth noting that the phrase "Network Adequacy" appears in studies of electric grids bearing a meaning that is NOT related to these distance algorithms. (Mahdavi and Mahdavi 2011; Ahmadi et al. 2019) Satellite "coverage" appears similar at first, and in some cases (like GPS or Satellite Internet) asks a similar question, but often the term "coverage" has a temporal component - for example with satellite imaging - where a satellite must pass over every point it wants to cover *at some point in time.* We do not explore this treatment for those problems with temporal components, although with some works the ideas may be extended there.

2 Introducing the Multilateration Index

2.1 Multilateration Index – General Definition

Given an n-dimensional metric space (M, d) comprising universe of points M in the space and a distance function $d(x, y)$ which respects the triangle inequality, a typical point X in the coordinate system will be described by coordinates $x_1, x_2, \ldots, x_n$, which, typically, represents the decomposition of a vector V from an "origin" point $O : 0, 0, \ldots, 0$ to X into orthogonal vectors $\{x1, 0, .., 0\}, \{0, x2, 0, .., 0\}, \ldots, \{0, 0, .., 0, xn\}$ along each of the n dimensional axes of the space.

The Multilateration of points in the space requires $n + 1$ fixed reference points F_p (p from 1 to $n + 1$), which contains no subset of any length m which all lie on the same $m - 2$ hyperplane. (i.e. in a 3d coordinate system, with four reference points, the four points cannot lie on the same plane, no three points can lie on the same line, and, trivially, no two points can be the same). The Multilateration Coordinate X' for the point X is then: $X' = \{t_1, t_2, \ldots, t_{n+1}\}$ where t_i is the distance $d(X, F_i)$ (in units applicable to the system).

A "Multilateration Index" is a data structure which, for a given set of points $p \in M$ stores the Multilateration Coordinates. Depending on implementation and use, these can be stored in a single structure sorted by one of t_i, or in multiple structures sorted individually by each t_i, or some other variant. See Multilateration NN Algorithms and Multilateration NA Algorithms for our specific implementations.

2.2 2-D Bounded Example

Consider a 2-dimensional grid – a flattened map, a video game map, or any mathematical $x - y$ coordinate grid with boundaries. WOLOG in this example consider the two-dimensional Euclidean space $M = \mathbb{R}^{\not{\in}}$ and bounded by $x, y \, \epsilon \, \{0..100\}$. Also, let us use the standard Euclidean distance function for d. This is, trivially, a valid metric space.

Since the space has dimension $n = 2$, we need 3 fixed points F_p. While the Geospatial example on Earth has a specific prescription for the fixed points, an arbitrary space does not. We therefore prescribe the following construction for bounded spaces:

Construct a circle (hypersphere for other dimensions) with the largest area inscribable in the space. In this example, that will be the circle centered at $(50, 50)$ with radius $r = 50$.

Select the point at which the circle touches the boundary at the first dimension (for spaces with uneven boundary ratios, select the point at which the circle touches the earliest boundary x_i). Such a point is guaranteed to exist since the circle is largest (if it does not, then the circle can be expanded since there is space between every point on the circle and an axis, and it is not a largest possible circle).

From this point, create a regular $n+1$-gon (triangle here) which touches the circle at $n+1$ points. These are the points we will use as F_p. They are, by construction, not all co-linear (or in general do not all exist on the same n-dimensional hyperplane) satisfying our requirement [proof].

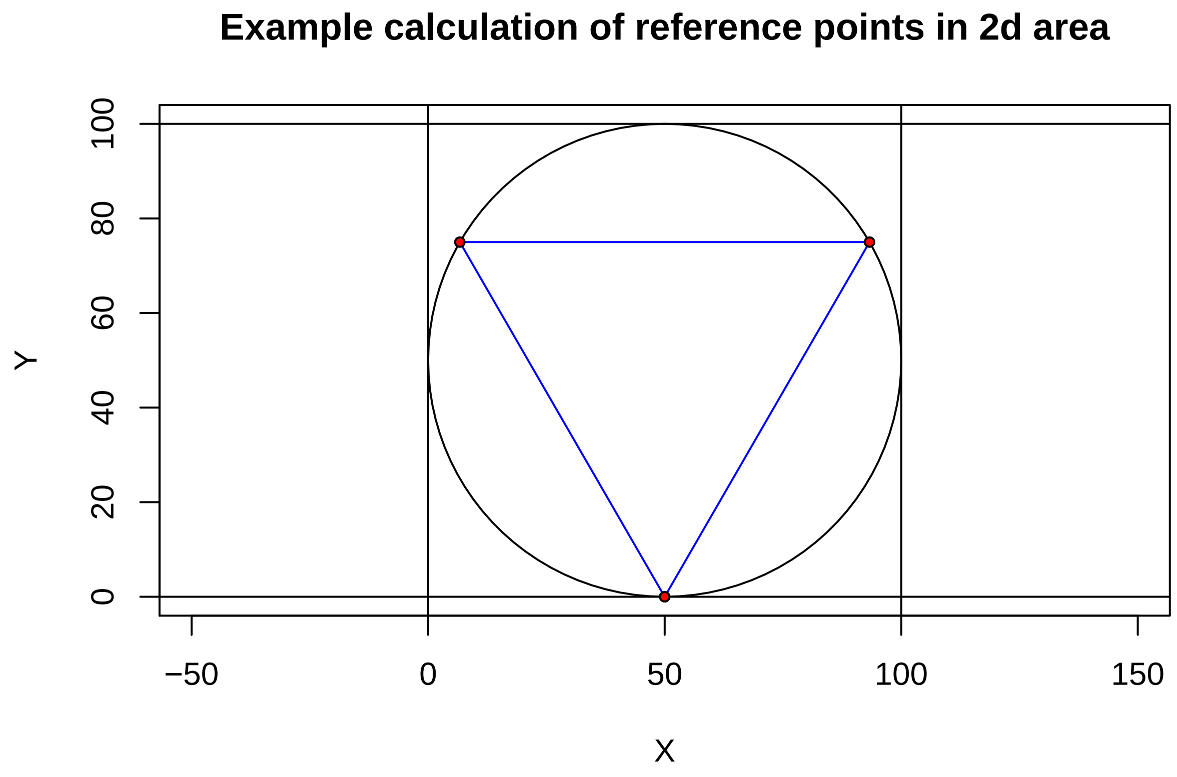
The point $y = 0$, $x = 50$ is the first point of the equilateral triangle. The slope of the triangle's line is $tan(\frac{pi}{3})$, so setting the equation of the circle:

$(x - 50)^2 + (y - 50)^2 = 50^2$ equal to the lines: $y = tan(\frac{\pi}{3})(x - 50)$ gives $x = 25(2 + \sqrt{3})$ on the right and $y = tan(\frac{-\pi}{3})(x - 50)$ gives $x = -25(\sqrt{3} - 2)$ on the left, and of course the original $(0, 50)$ point. Applying x to our earlier equations for y we get a final set of three points:

$$F_1 = (x = 50, y = 0)$$

$$F_2 = (x = 25(2 + \sqrt{3}), y = tan(\frac{\pi}{3})((25(2 + \sqrt{3})) - 50)$$

$$F_3 = (x = -25(\sqrt{3} - 2), y = tan(\frac{-\pi}{3})((-25(\sqrt{3} - 2)) - 50)$$



Remember, any three non-colinear points will do, but this construction spaces them fairly evenly throughout the space, which may be beneficial later* [Add section (reference) with discussions of precision and examples where reference points are very near one another].

The trilateration of any given point X in the space, now, is given by:

$$T(X) = d(F_1, X), d(F_2, X), d(F_3, X)$$

That is, the set of (three) distances d from X to F_1, F_2, and F_3 respectively.

2.2.0.1 10 Random Points

As a quick example of the trilateration calculations, we use a basic collection of 10 data points:

Table 1: 10 Random Points

x	y
58.52396	53.516719
43.73940	43.418684
57.28944	7.950161
35.32139	58.321179
86.12714	52.201894
41.08036	78.065907

x	y
51.14533	47.157734
15.42852	80.836340
85.13531	64.090063
99.60833	78.055071

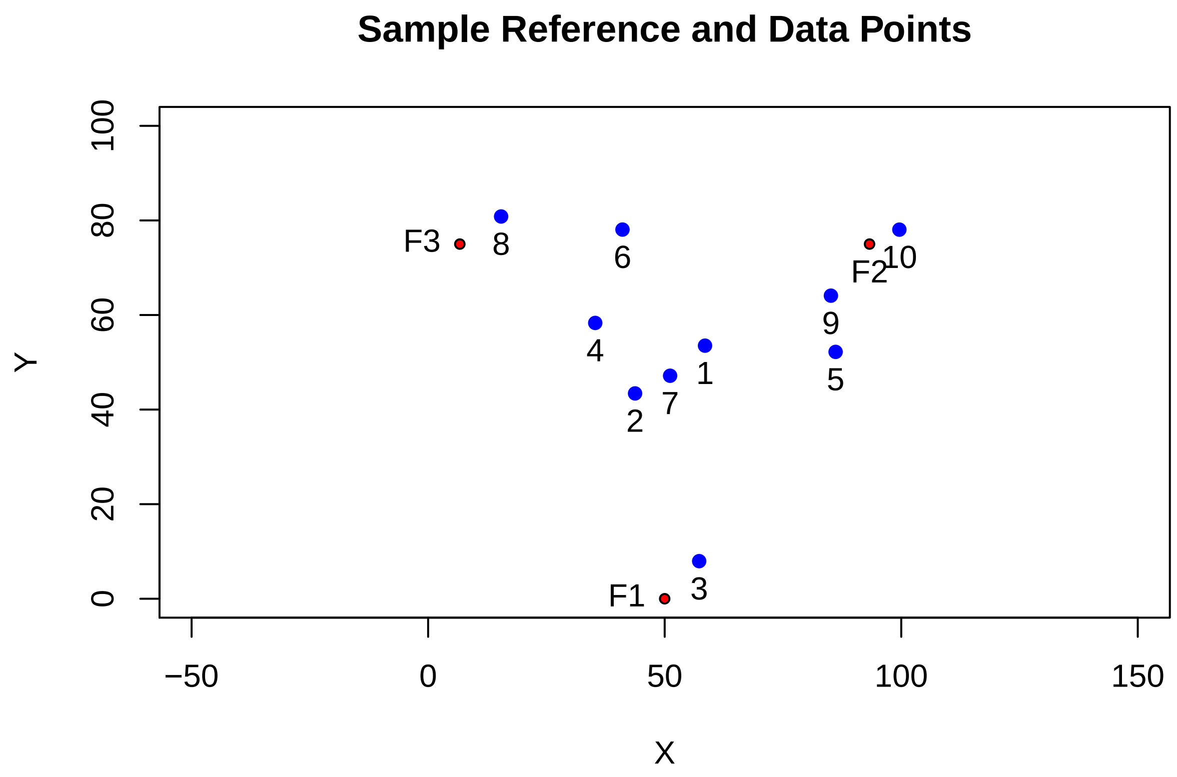
The trilateration of those points, that is, the three points $d_1, d_2, d_3 = d(F_1, X), d(F_2, X), d(F_3, X)$ are (next to the respective x_n):

Table 2: Multilateration Index of 10 Random Points WRT Fixed points F

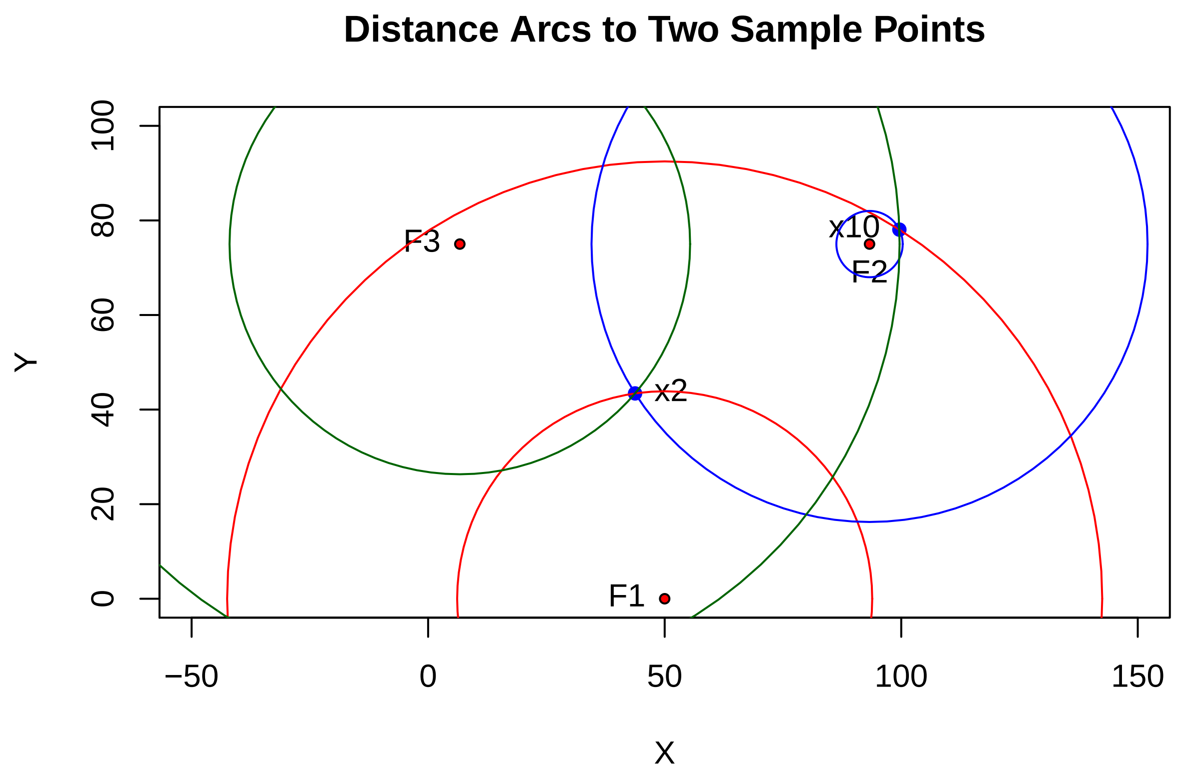
x	y	d1	d2	d3
58.52396	53.516719	54.19130	40.877779	56.10157
43.73940	43.418684	43.86772	58.768687	48.67639
57.28944	7.950161	10.78615	76.108693	83.99465
35.32139	58.321179	60.14001	60.331164	33.12763
86.12714	52.201894	63.48392	23.900247	82.63550
41.08036	78.065907	78.57382	52.310835	34.51806
51.14533	47.157734	47.17164	50.520446	52.44704
15.42852	80.836340	87.91872	78.091150	10.50105
85.13531	64.090063	73.08916	13.627535	79.19169
99.60833	78.055071	92.48557	7.008032	92.95982

Note that we do not need to continue to store the original latitude and longitude. We can convert the three d_n distances back to Latitude and Longitude within some ϵ based on the available precision. Geospatial coordinates in Latitude and Longitude with six digits of precision are accurate to within < 1 *meter*, and 8 digits is accuract to within < 1 *centimeter*, although this varies based on the latitude and longitude itself; latitudes closer to the equator are less accurate than those at the poles. The distance values d_x are more predictable, since they measure distances directly. While the units in this sample are arbitrary, $F(x)$ in a real geospatial example could be in kilometers, so three decimal digits would precisely relate to 1 *meter*, and so on. This is one reason that we will later examine using the trilateration values as an outright replacement for Longitude and Latitide, and this feature is important when considering storage requirements for this data in large real-world database applications.

For now, continuing with the example, those 10 points are shown here in blue with the three reference points F_1, F_2, F_3 in red:



To help understand the above values, the following chart shows the distances for points x_2 and x_10 above. Specifically, the distances d_1 from point F_1 are shown as arcs in red, the distances d_2 from point F_2 in blue, and d_3 from point F_3 in green.



2.2.1 Geospatial Example

Applying this to real sample points; let the following be the initial reference points on the globe:

Point 1: $90.000000, 0.000000$ (The geographic north pole)

Point 2: $38.260000, -85.760000$ (Louisville, KY on the Ohio River)

Point 3: $-19.22000, 159.93000$ (Sandy Island, New Caledonia)

Optional Point 4: $-9.42000, 46.33000$ (Aldabra)

Optional Point 5: $-48.87000, -123.39000$ (Point Nemo)

Note that the reference points are defined precisely, as exact latitude and longitude to stated decimals (all remaining decimal points are 0). This is to avoid confusion, and why the derivation of the points is immaterial (Point Nemo, for example is actually at a nearby location requiring more than two digits of precision).

Only three points are required for trilateration (literally; thus the "tri" prefix of the term), but we include 5 points to explore the pros and cons of n-fold geodistance indexing for higher values of n.

2.3 Theoretical Discussion

2.3.1 Theoretical benefits:

Precision: Queries are not constrained by precision choices dictated by the index, as can be the case in Grid Indexes and similar R-tree indexes. R-tree indexes improve upon naïve Grid Indexes in this area, by allowing the data to dictate the size of individual grid elements, and even Grid Indexes are normally tunable to specific data requirements. Still, this involves analysis of the data ahead of time for optimal sizing, and causes resistance to changes in the data.

Distributed Computing: Trilateration distances can be used as hash values, compatible with distributed computing (I.e. MongoDB shards or Teradata AMP Hashes).

Geohashing: Trilateration distances can be used as the basis for Geohashes, which improve somewhat on Latitude/Longitude geohashes in that distances between similar geohashes are more consistent in their proximity.

Bounding Bands: The intersection of Bounding Bands create effective metaphors to bounding boxes, without having to artificially nest or constrain them, nor build them in advance.

Readily Indexed (B-Tree compatible): Trilateration distances can be stored in traditional B-Tree indexes, rather than R-tree indexes, which can improve the sorting, merging, updating, and other functions performed on the data.

Fault Tolerant: This coordinate system is somewhat self-checking, in that many sets of coordinates that are individually within the correct bounds, cannot be real, and can therefore be identified as data quality issues. For example, a point cannot be 5 kilometers from the north pole (fixed point F1) and 5 kilometers from Louisville, KY (fixed point F2) at the same time. A point stored with those distances could be easily identified as invalid.

Theoretical shortcomings:

Index Build Cost: Up front calculation of each trilateration is expensive, when translating from standard coordinates. Each point requires three (at least) distance calculations from fixed points and the sorting of the resulting three lists of distances. This results in `O(n*logn)` just to set up the index.

*This could be mitigated by upgrading sensor devices and pushing the calculations back to the data acquisition step, in much the way that Latitude and Longitude are now trivial to calculate in practice by use of GPS devices. Also, we briefly discuss how GPS direct measurements (prior to converstion to Lat/Long) may be useful in constructing trilateration values.

Storage: The storing of three distances (32- or 64- bits per distance) is potentially a sizeable percent increase in storage requirement from storing only Latitude/Longitude and some R-Tree or similar index structure.

*Note that if the distances are stored instead of the Lat/Long, rather than in addition to them, storage need not increase.

Projection-Bound: The up-front distance calculations means that transforming from one spatial reference system (I.e. map projection – geodetic – get references to be specific) to another requires costly recalculations bearing no benefit from the calculation. For example a distance on a spherical projection of the earth between a given lat/long combination will be different than the distance calculated on the earth according to the standard WGS84 calculations).

*This said, we expect in most real-world situations, cross-geodetic comparisons are rare.

Difficult Bounding Band Intersection: Bounding Bands intersect in odd shapes, which, particularly on ellipsoids, but even on 2D grids, are difficult to describe mathematically. Bounding boxes on the other hand, while they distort on ellipsoids, are still easily understandable as rectangles.

2.4 Underlying Theory Concepts and Problem Statement

The benefit of storing geographic points as a set of trilateration distances rather than latitude and longitude boils down to the simplification of comparing distances between points by shortcutting complex distance queries using simple subtractions. We discuss the math behind the geospatial queries, to exhibit their complexity, and set some theoretical bounds on quick distance calculations using the trilateration index.

2.4.1 Problem: High Cost of Geospatial Calculations

Calculating the distance between two points around the globe with precision is required for Satellite Communications and Geospatial Positioning Systems (GPS), as well as for ground based surveying and generally all applications requiring precise (sub-meter) measurements accounting for the curvature of the earth.(ASPRS 2015) Recall from our research that modern methods of calculating accurate distances on a properly modeled (elliptical) earth requires calculating a converging series with an iterative algorithm. This is significantly more computationally expensive than closed form mathematical distance functions like the Haversine spherical distance or more common Euclidean distances.

To get an idea of the relative complexity, we ran some basic timings using widely available python libraries that perform both calculations. The Haversine is about 22 times faster than Karney's iterative approach. For comparison, we include Euclidean functions, which are of course computationally simple, although their usefulness on curved surfaces are minimal, and subtraction, which our algorithms use as a bounding function in place of exact results, thus leveraging this timing delta:

Table 3: Timings (seconds) of 5000 Calls to Distance Functions

title	time	ratio
Geodesic	1.2110690	575.548427
Haversine	0.0553729	26.315417
Euclidean	0.0021042	1.000000
Subtraction	0.0012876	0.611919

2.5 Simple Multilateration Index Operations

Before jumping into Network Adequacy and Nearest Neighbor algorithms let's look at the core usage of the trilateration data structure and its use in simple distance functions.

What we mean by 'simple distance functions' is one of the following primitive functions common to SQL or map related software libraries:

- $D(p, q)$: returns the distance between points p and q
- $Within(d, q, P)$: returns the set of all points in P within distance d of query point q ** $CountWithin(d, q, P)$: returns the count of the set of points from $Within(d, q, P)$
- $AnyWithin(d, q, P)$: returns a boolean result - True if $Within(d, q, P)$ is non-empty; False otherwise

2.5.1 Distance Function

How can we use the Trilateration Index (TI) to improve the performance of a single distance function $D(p, q)$? In the simplest case, we cannot… the construction of the TI structures requires three distance functions to be calculated each for p and q (to the three fixed reference points).

However, for large datasets with fixed points where many distances need to be calculated between them, particularly if the distance function itself is computationally intensive (such as geospatial distances on an accurate ellipsoid model of earth) (Lambert 1942), we can use the TI structure to create approximate distances, and provide upper and lower bounds on exact values.

For example, let's take our sample data:

	x	y	d1	d2	d3
8	15.42852	80.836340	11.63066	78.32131	88.19694
4	35.32139	58.321179	33.14272	60.36431	60.21479
6	41.08036	78.065907	34.64818	52.46354	78.73275
2	43.73940	43.418684	48.68666	58.76869	43.87912
7	51.14533	47.157734	52.59935	50.76727	47.55169
1	58.52396	53.516719	56.13720	40.89001	54.19130
9	85.13531	64.090063	79.41866	15.32024	73.52568
5	86.12714	52.201894	82.65970	24.08779	63.60981
3	57.28944	7.950161	83.99465	76.11526	10.97000
10	99.60833	78.055071	93.22300	10.63544	92.92245

Here, X and Y are euclidean cartesian coordinates, and d1, d2, d3 are the distances from these points to our three reference points respectively. See 2-D Bounded Example for more details on the construction. Note that in this case we have sorted the data by $d1$ – this is essential, and incurs only $O(n * log(n))$ overhead. This equates to how database indexes or arrays will hold the data in memory.

2.5.2 Distance between two points

If we compare points 1 and 2 here (lines 4 and 2 in the $d1$-sorted table), what can we say about those two points' distances without invoking a distance function? If we compare the distances, we can put lower bounds on their proximity using a direct, simple application of the triangle inequality. For example $|d1(P_1)-d1(P_2)| = |54.19130-43.87912| = 10.33$ which means the points can be ***no closer than*** 10.33 units to one another. Similarly with d2 and d3, we get $|58.76869-40.89001| = 17.88$ and $56.13720-48.68666 = 7.45054$. So now, the points can be no closer than 17.88 units, although they are closer relative to the $d1$ and $d3$ points.

2.5.3 Within/AnyWithin Distance

It's similarly easy to use this mechanism to approximate answers to "which points are within distance d of query point Q?" and, relatedly, "is there at least one point in P within distance d to point Q?".

Looking back at our table, let's examine the question "which points are within distance 20 of point 5?". Point 5 has coordinates $(86.12714, 52.201894)$, and is 63.60981 units from $d1$. Since we've stored the list sorted by $d1$, we can instantly limit our search to a sequential walk from points between 43.60981 and 83.60981 – that is, points $(7, 1, 4, 9, 6)$ (excluding 5 itself). This is, immediately, a 50% reduction in the dataset.

While performing the walk, we look for $d2$ between 24.08779 ± 20 and $d3$ between 82.65970 ± 20. $d2$ rules out points $(7, 4, 6)$ and $d3$ rules out (1), leaving only (9) for consideration. To be completely certain, we can calculate $d = \sqrt{(86.12714 - 85.13531)^2 + (64.090063 - 78.065907)^2} = 14.0109936$ which is, indeed, within 20.

In pseudocode:

```
Within(d, P, TI):
   lowi = lowest i such that TI[i, d1] > P[d1] - d
   highi = highest i such that TI[i, d1] < P[d1] + d
   FOR i FROM lowi to highi:
       if TI[i, dx] between P[dx] - d and P[dx] + d for all x:
```

```
6           ADD i to CANDIDATES
7      FOR c in CANDIDATES:
8          if D(c, P) < d:
9             ADD c to RESULTS
10     RETURN RESULTS
```

CountWithin is simply the same code but returning the count of *RESULTS* not the points themselves.

If we were answering the "is there at least one point…" *AnyWithin* version, it would be easy to shortcut the sequential walk when a match is reached.

2.5.4 Alternate Order Indexes

For an additional possible performance improvement, we can create alternate indexes which store the data in sorted order along $d2$ and $d3$ (or any/all distances for arbitarty dimensions). We search for the low and high indexes as before, but now we do so along each sorted index (for distances to each reference point). Once we have the lists of individual candidates from each index, we need to find any point that is common to all candidate lists. In practice we have not seen this behave as effectively as the single-index function, but this seems to come down to the cost of merging n-lists to find common elements.

In pseudocode:

```
WithinMulti(d, P, TI):
   FOR each ref point rx:
      lowi = lowest i such that TI[i, dx] > P[dx] - d
      highi = highest i such that TI[i, dx] < P[dx] + d
      FOR i FROM lowi to highi:
          if TI[i, dx] between P[dx] - d and P[dx] + d:
             ADD i to CANDIDATES[x]
    FOR c in CANDIDATES[1]:
      if c in CANDIDATES[x] for all x:
          ADD c to POSSIBLE
    FOR c in POSSIBLE:
       if D(c, P) < d:
          ADD c to RESULTS
    RETURN RESULTS
```

2.5.5 Time Complexity

2.5.5.1 Estimation of Time Complexity of "Within" query

Note that lines 2 and 3 are $O(log_2(n))$ operations, since we can do a binary search on the sorted TI structure to find points closest to a specific value. The loop in line 4 is a sequential walk along the array; the time complexity being subject to the distance d and the composition of the points P.

To estimate time complexity in random or average cases, let's take a closer look at what's happening with some visual elements. Looking at the "Within Complexity" Figure, we see the following:

- Three reference points $r_1..r_3$ as triangles
- 10 data points $p_1..p_{10}$ as numbered small circles
- The query point q, a small square
- A dashed circle with radius d. Note that, in this example, point 2 lies precisely d units from q, with $d \approx 22.47$
- Dashed lines from r_1 to each point indicating their distance p_{d1} (and sort order in TI) from r_1
- A large ring representing all of the space where a point within d of q must lie, with respect to r_1

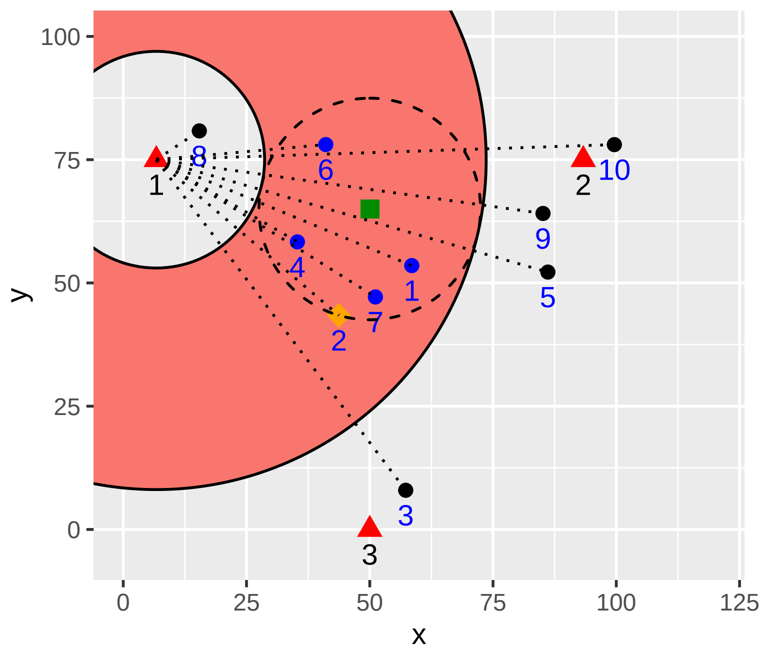


Figure 1: Within time complexity illustration

Comparing this to our sample data, the length of the dashed lines correspond to the sort order in the TI index. That is, $8, 4, 6, 2, 7, 1, 9, 5, 3, 10$ are increasingly distant from r_1. This is how we quickly prune the list – points within d distance of q are, by definition, between $q_{d1} - d$ and $q_{d1} + d$ distance from r_1 - namely, points $4, 6, 2, 7, 1$. The rest can be no closer than this. In this case, all the points in the ring happen to also be within d of q, but of course that will not always be the case.

Clearly there are antagonistic datasets that would thwart any benefit. If the data were arranged in a circle all near distance d from r_1, we would have culled no points from this approach. In that particular case, the alternate $WithinMulti$ construction would be beneficial.

If the points were randomly distributed, however, then the expected culling ratio of the index becomes the area of the ring divided by the area of the total point space. On an infinite plane, of course, this approaches zero, but in finite spaces, such as this example (and geospatial coordinates), we can calculate the area. Here, we use a common Monte Carlo method to estimate that, for this distance, on the $100x100$ grid, the ring covers about 48.7% of the area.(Wasserstein, Kalos, and Whitlock 1989) Given that we had to check five of our ten points, this seems reasonable.

2.5.5.2 Time complexity of WithinMulti

If, rather than cull by one ref point then search the remainder, we cull by all three (the *WithinMulti* approach), we see something different:

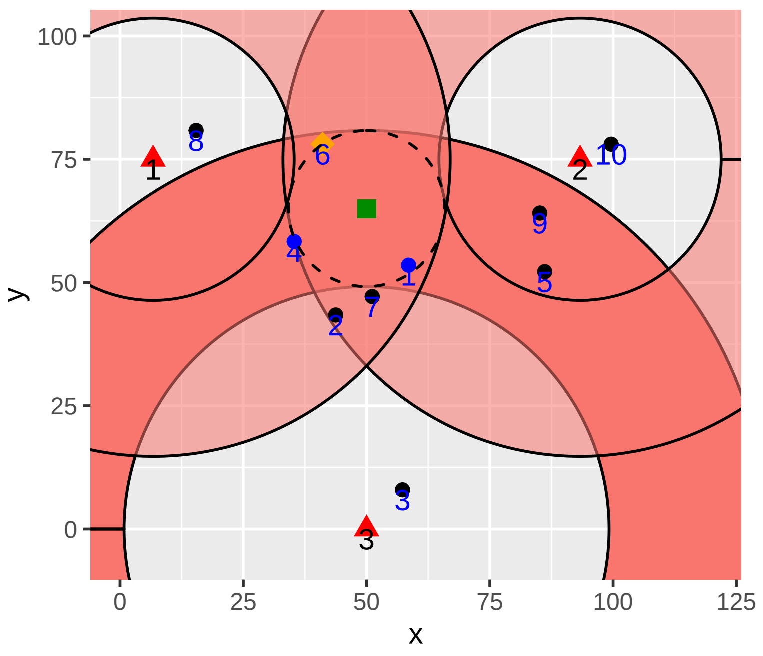


Figure 2: WithinMulti time complexity illustration

This time we've used a slightly smaller d - the distance from q to p_6 or $d \approx 16.127$, which helps illustrate excluded points $2, 7$.

To perform this efficiently, we would need to store additional sorted lists in memory - namely the points P sorted along distances from r_2 and r_3 respectively… for example the index with respect to r_2 could look like this (x and y need not even be stored again to save more memory, as long as the point index is available):

Table 5: Indexed sample data with respect to r2

	x	y	d2
10	99.60833	78.055071	10.63544
9	85.13531	64.090063	15.32024
5	86.12714	52.201894	24.08779
1	58.52396	53.516719	40.89001
7	51.14533	47.157734	50.76727
6	41.08036	78.065907	52.46354
2	43.73940	43.418684	58.76869
4	35.32139	58.321179	60.36431
3	57.28944	7.950161	76.11526
8	15.42852	80.836340	78.32131

Here, culling points along the three distances respectively we get:

- r_1: $4, 6, 2, 7, 1$
- r_2: $1, 7, 6, 2, 4$
- r_3: $1, 4, 6$

It's clear that r_3 is the most restrictive; examining the images, it's easy to see why… points 2 and 7 reside within the rings aroudn r_1 and r_2, but not that of r_3. The final step, of finding which elements are common to these three lists is, unfortunately, not efficient. We can start with the smallest list, which is some help, but as these are sorted in different orders, finding the intersection of all of the lists is, algorithimcally, non-trivial. Given m sets containing N elements total which result in o points in the intersection, modern research indicates space complexity of at least $O(N \log(N))$ and time complexity $O(\sqrt{N * o} + o)$).(Cohen and Porat 2010)

Still, the improvement in the initial cull may be worth it. Recall that, using a single reference point, the time complexity was a result of the area of the ring divided by the area of the space. In the multi-reference-point version, with m reference points, the result is m times the area of the intersection of the m rings, divided by the area of the space.

2.5.5.3 Numerical Approximation:

In this current example, with $d \approx 16.127$, we can numerically approximate this value:

- total area: $100x100 = 10000$
- r_1 ring area is $\approx 33.1\%$
- r_2 ring area is $\approx 32.7\%$
- r_3 ring area is $\approx 37.9\%$
- Intersection of all three rings has area $\approx 9.4\%$

A Monte Carlo simulation of this is illustrated in Figure: "Monte Carlo Estimating Ring Overlap Area".

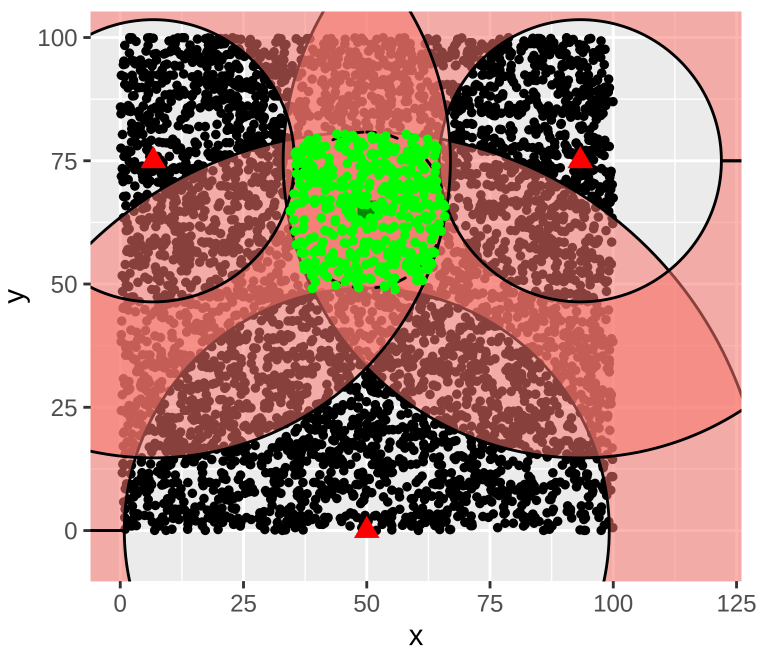


Figure 3: Monte Carlo Estimating Ring Overlap Area

Again, these numbers will vary with d more than anything else, but the concept here is that for low d, the area of the ring can be significantly less than that of the entire search area, even moreso the intersection of the area of three such rings, benefitting the actual time complexity of the query.

3 Experimentation

3.1 Experimental Setup

We plan experiments for Network Adequacy in SQL databases, and Nearest Neighbor applications using Python.

3.1.1 Experiment 1: Python Nearest Neighbors with ANN

1. We adapt the popular Python package scikit-learn to execute our Trilateration Index and perform nearest-neighbor search.
2. We adapt the ann-benchmarks software, which is designed explicitly for comparing performance of nearest neighbor algorithms(Aumüller, Bernhardsson, and Faithfull 2020), to record results. This includes the addition of a geodesic dataset and distance function (ann-benchmarks only supported jaccard, hamming, euclidean, and angular distance functions) for consistency (see [Geodesic query points] for details).
3. We implement the four algorithms described in Multilateration NN Algorithms, namely "Trilateration" (TRI), "TrilaterationApprox" (TIA), "TrilaterationExpand" (TIE), and "TrilaterationExpand2" (TIE2)
4. We execute the ann-benchmarks software against our and other modern NN algorithms, and record the results in [ANN Benchmarks]

Note that we use a widely available geodesic distance implementation from Geopy to provide a consistent distance function implementation, thus avoiding any bias in results due to differences between implementations.(Brian Beck, n.d.) Since the geodesic distance function is the result of a convergent series, it is possible to vary the precision of the calculation, trading performance for accuracy. We use the default settings in the Geopy implementation, again for a consistent comparison across NN techniques.

3.1.2 Experiment 2: SQL Network Adequacy

For Network Adequacy, no standard benchmark exists (such as ann-benchmarks for Nearest Neighbor), so our experimental setup requires a bit more setup. We use the following steps:

1. We take the same 150,000 point geospatial data set used in the NN experiment (see [Geodesic query points] for details) and assign the points randomly to 15 categories of varying sizes (see table [Record Counts by Category ID]).

Table 6: Record Counts by Category ID

	0-9	10	11	12	13	14	15
Record Count	10000	100	900	2000	5000	10000	32000

2. We implement our two Network Algorithms (NAIVE-NA and TRILAT-NA) described in Multilateration NA Algorithms in the SQL database.
3. We execute these over various combinations of categories in our dataset, logging the duration of each combination for comparison.

3.2 Experimental Results

3.2.1 Nearest Neighbor Results

3.3 Results of Experiment 1 - ANN Benchmarks

3.4 Geodesic query points:

We have created a dataset specifically to test Geodesic queries. The dataset is a synthetic set of 150,000 geospatial points spread across and near Kentucky with roughly the distribution of the population.

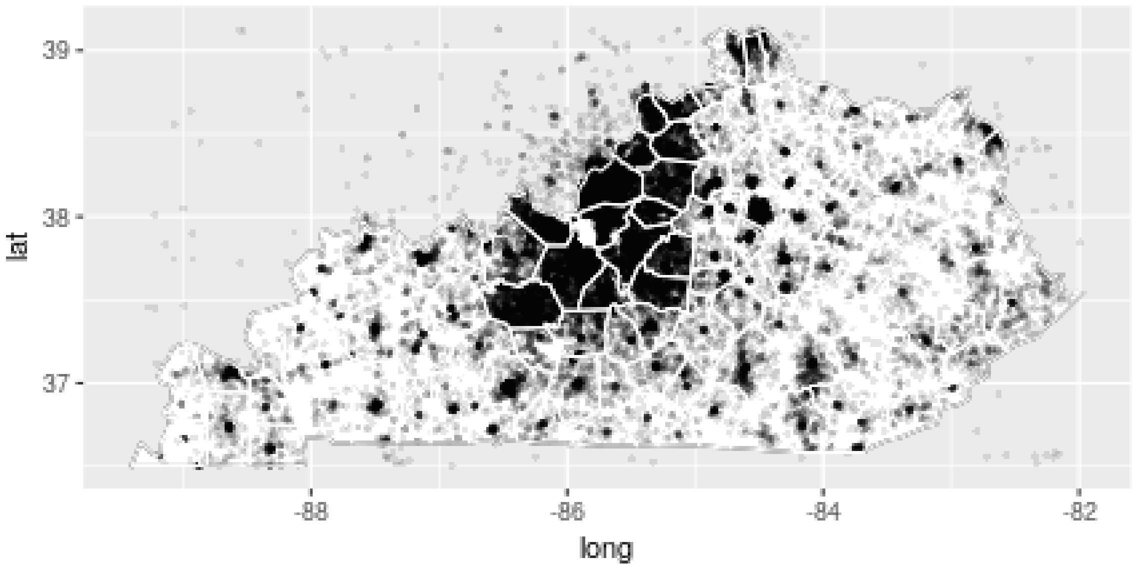


Figure 4: Geodesic Sample Data

As hoped, our Trilateration algorithm really shines when applying the complex but accurate geodesic distance function. The Trilateration algorithm is over 30 times faster than the next best candidate (the Ball Tree algorithm with leaf_size=10). The Brute Force algorithm is unbearably slow here, which is expected since it should be calling the expensive distance function n^2 or $150,000^2 = 22,500,000,000$ (22.5 billion) times... more than any other algorithm by far. Recall that our test of 5000 calls to a single geodetic function took $\approx$ 1.2 seconds, so we'd expect 22.5 billion to take 5.4 million seconds, or about 62 days on similar hardware. The other algorithms ran to completion.

3.5 Random 20-dimension euclidean distance:

The ann-benchmark tool includes support for a randomly generated 20-dimension euclidean dataset, which is one of its most basic tests. We note that our performance here is abysmal for the initial "Trilateration" algorithm, even failing to beat the brute force approach. This seems to be due to the overhead we incur determining which candidates to test next. Remember that we traded a number of subtractions and some

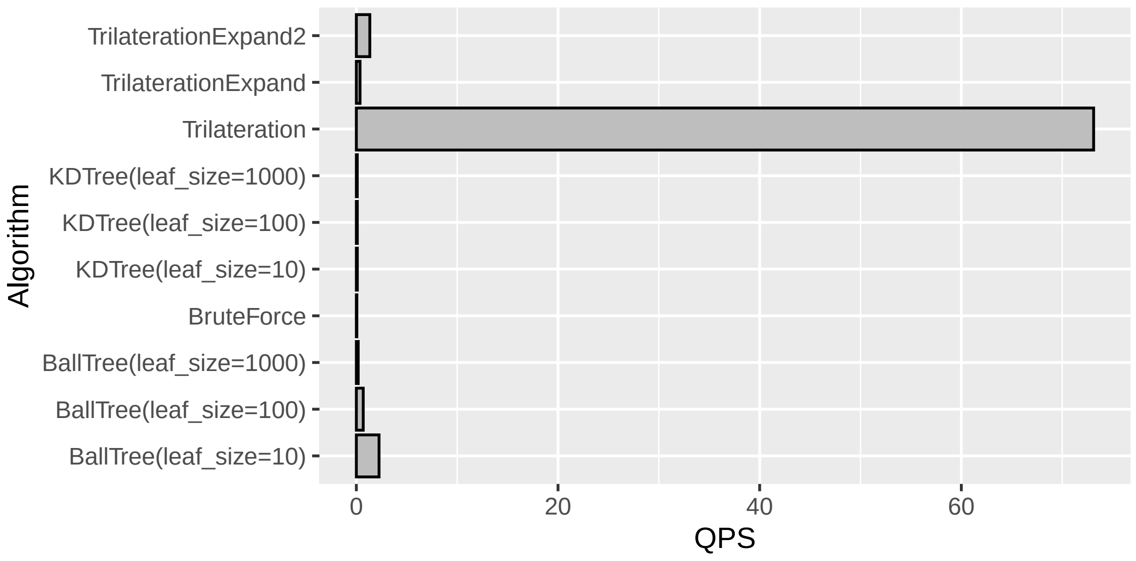


Figure 5: Queries Per Second for Geodesic (Q=150,000; higher is better)

array navigation in exchange for fewer distance function calls. In this case, when the distance function itself is extremely fast, that overhead is a net loss.

It is worth noting that we tuned the two expansion based algorithms here as well. "TrialterationExpand" performs far better than the stock Trilateration algorithm, but still just below the Brute Force algorithm. The "TrilaterationExpand2" algorithm, however, is actually competitive here, more than doubling the queries per second of the Brute Force approach, and reaching more than 60% as fast as some tree algorithms.

For Euclidean distances, however, we cannot recommend our algorithms against competitors.

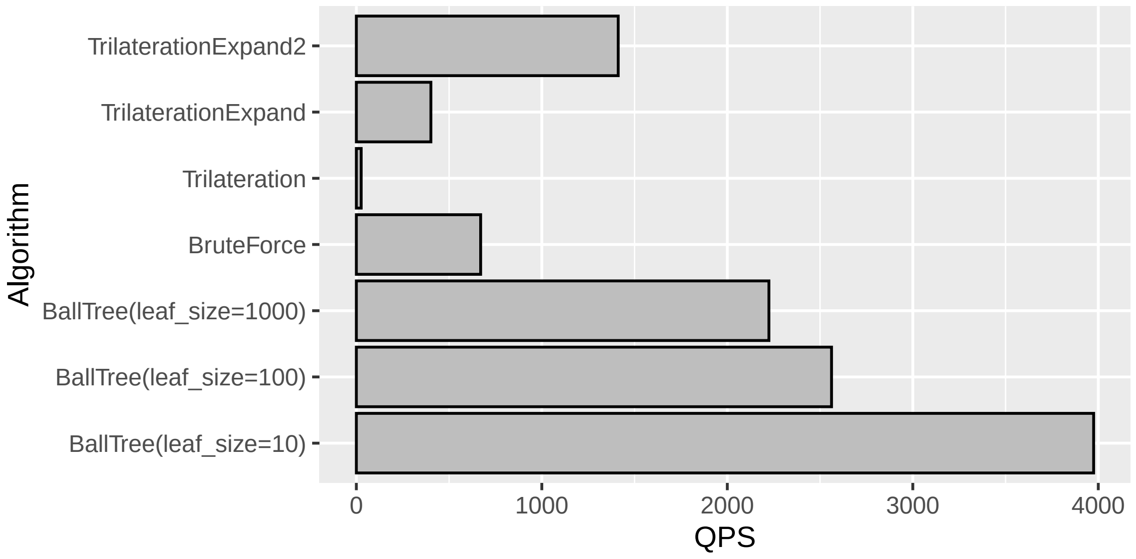


Figure 6: Queries Per Second and Time to Build Indexes for 20-dimension Euclidean

3.6 Glove 25-dimension Angular distance

The GloVe ("Global Vectors for Word Representation") dataset "is an unsupervised learning algorithm for obtaining vector representations for words. Training is performed on aggregated global word-word co-occurrence statistics from a corpus, and the resulting representations showcase interesting linear substructures of the word vector space" according to the authors. (Pennington, Socher, and Manning 2014)

It is one of the built-in datasets in the ann-benchmark tool. Under the hood, this isusing a euclidean distance function, once the angular coordinates are transformed, so the relative performance is similar to the euclidean dataset.

Of note, we include only results that were full (not approximate) nearest neighbor solutions.

These results include the addition of the FAISS algorithm, one of the graph index based NN solvers which came out of Facebook Research in recent years.(Johnson, Douze, and Jégou 2017) On these datasets, FAISS is a beast, but unfortunately it is a very highly tuned GPU-based implementation which makes it difficult to adapt to unsupported distance functions, such as the Geodesic we target with Trilateration. We leave it here for information, but are unable to compare it on our core task.

Note that our performance has suffered again similarly to with the random euclidean dataset. Trilateration is extremely slow; as is TrilaterationExpand. TrilaterationExpand2 beats BruteForce and is somewhat shy of the Tree-based algorithms. But we are not competitive in this space.

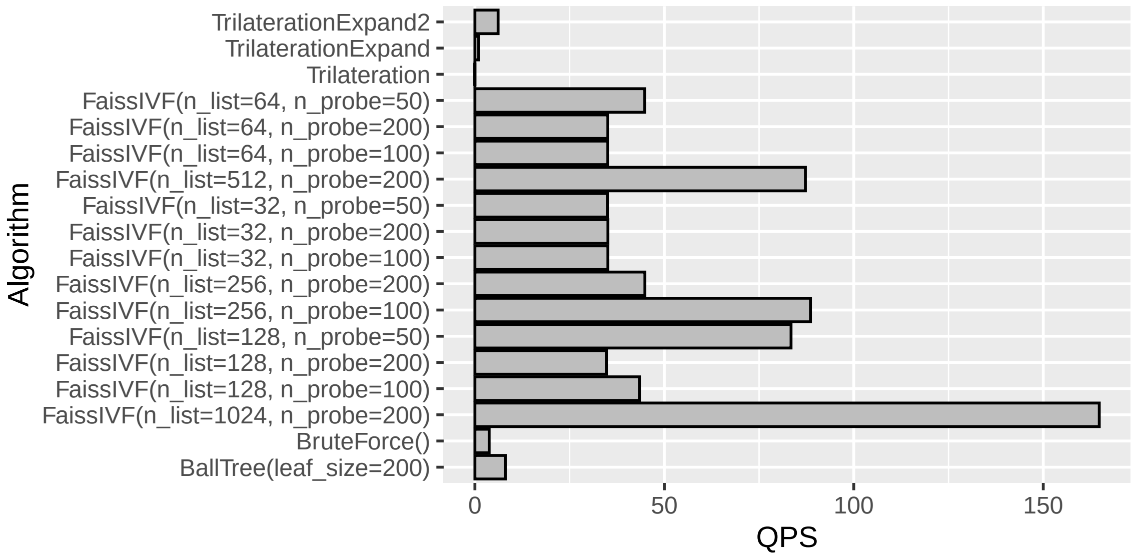


Figure 7: Queries Per Second and Time to Build Indexes for glove-25-angular

3.6.1 Network Adequacy Results

3.7 SQL Network Adequacy Results

We conducted our experiment as described, resulting in 36 combinations of point (P) and query (Q) sets from the same 150,000 point sample we've used for NN experiments. The datasets are categorized and range from 100 to 32,000 points, as shown here:

Table 7: Record counts by category used in experiment

	category	record count
166	10	100
168	11	900
180	12	2000
192	13	5000
276	14	10000
294	15	32000

Recall that we are calculating the Network Adequacy Percent (NAP), which requires a distance d, and we performed expriments with $d \in (0.1, 1, 10, 100)$. At a high level, our TRILAT-NA algoritm performs well against the naive algorithm as shown in our "SQL Timings" figure.

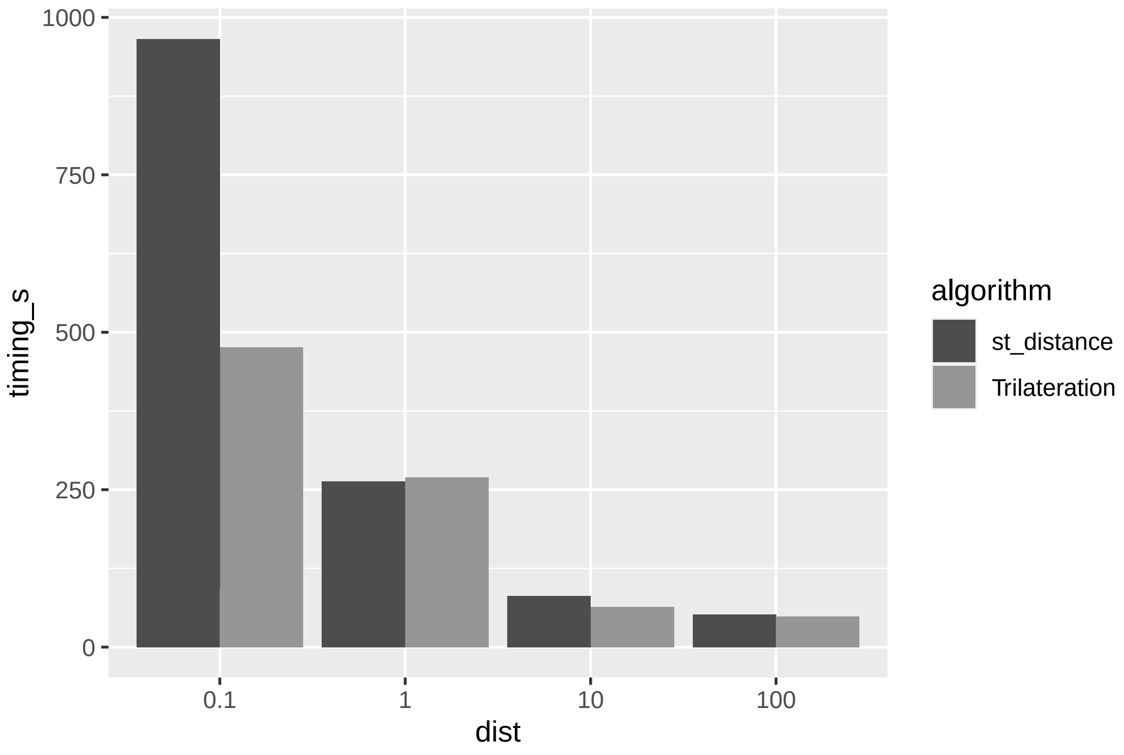


Figure 8: SQL Timings (s) - lower is better

Given the importance of the sizes of the P and Q datasets, it's quite informative to compare the algorithms with respect to those sizes. Also very important is the value of d. It's worth noting that the actual NAP for d varies in this dataset; using $|P| = 32000$ and $|Q| = 10000$ we find:

Table 8: d vs. ~NAP

d	NAP
0.1	0.42
1.0	0.88
10.0	0.98
100.0	1.00

Of course this varies with the size of the dataset, but the rough bounds are important. The data set comprises points in Kentucky which is roughly 400 miles east-west and 200 miles north-south. When d is 100 miles, even with 100 points (our smallest sample), 100% coverage is all but guaranteed. At 10 miles, we see some holes in coverage, but not many – thus the 98.4% value. When we get down to the one mile and one tenth of a mile ranges, coverages drop precipitously. Even with our largest two categories, NAP is only 88% at 1 mile and 42% at 0.1 miles. If we had been using real world data, imagine that we ask the question "what percent of people live within 1 mile of a pharmacy" to get an idea what these numbers could mean (if this were not synthetic data).

3.7.1 Result Charts

In any case, comparing performance those are our parameters... the size of Q, size of P, and the distance d. We chart the results here, for both the NAIVE-NA and TRILAT-NA queries:

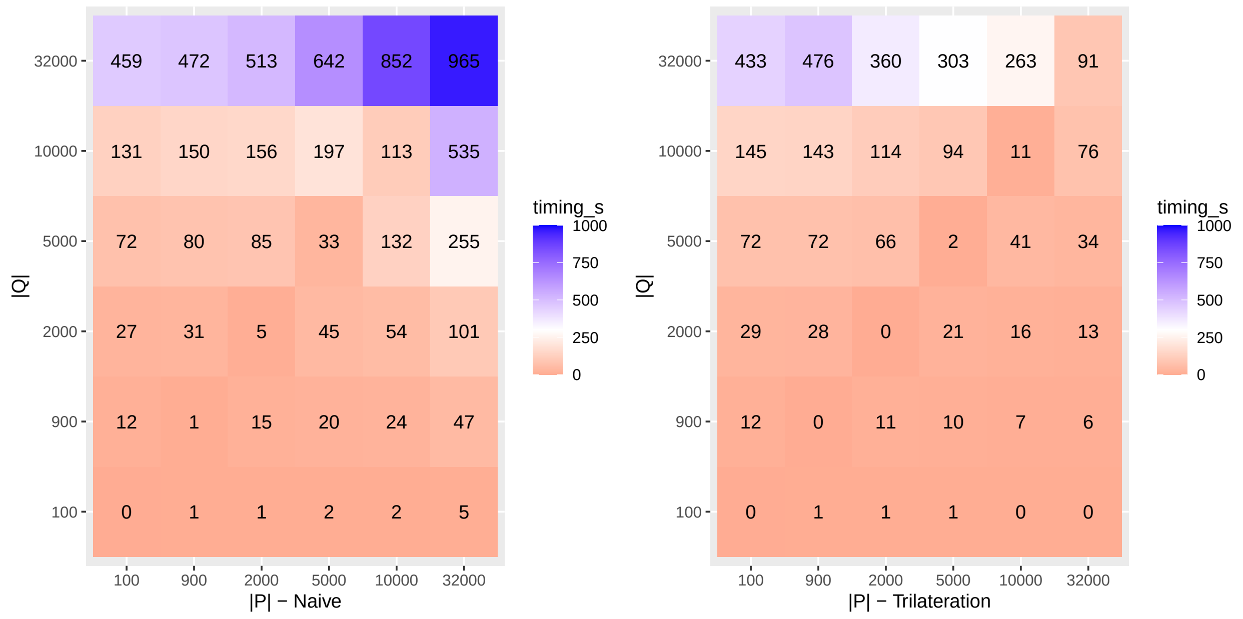


Figure 9: SQL Timings By Category (s) - d=0.1 - lower is better

---

3.7.2 Interpretation

We find these charts quite interesting. What is happening? we make several distinct observations:

3.7.2.1 Processing time is directly realted to |Q|

As the size of the query data set (Q) increases along the y-axis, it's clear that the computation time increases at roughly the same rate. The top row is roughly 3x the row below it (the ratio of 32,000:10,000)... the next row is roughly half (10,000:5000), and so on. This makes sense; if we consider the query a for loop over Q, which is how we implement it in another language, the cost should go up proportionately, and it clearly does.

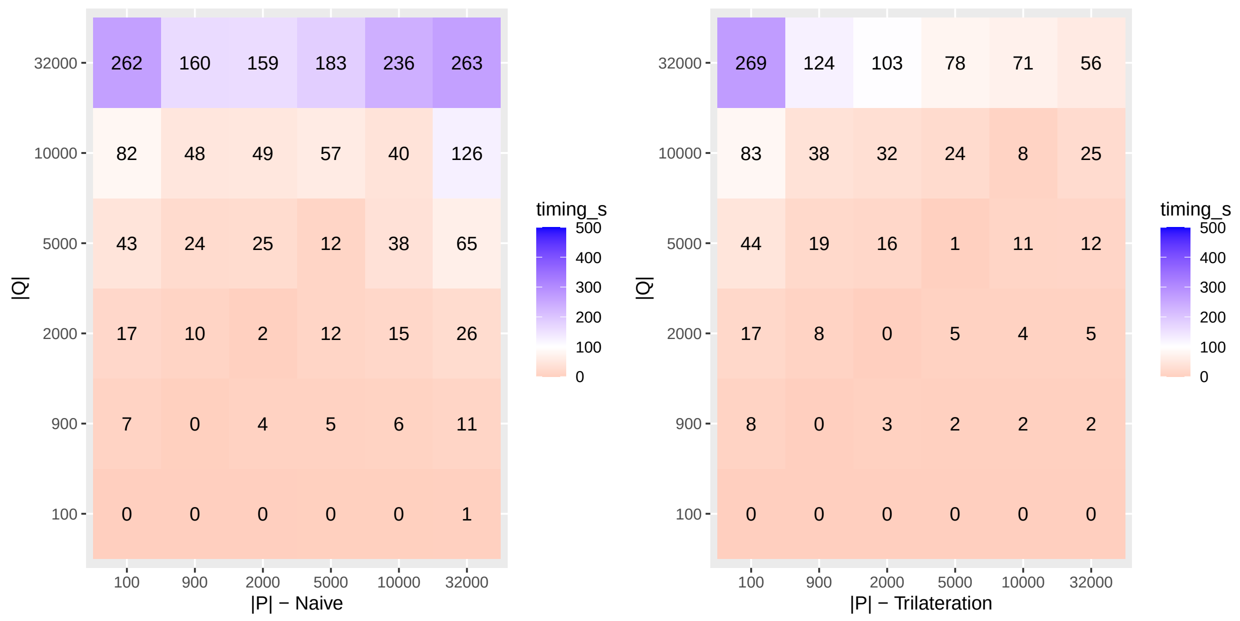


Figure 10: SQL Timings By Category (s) - d=1 - lower is better

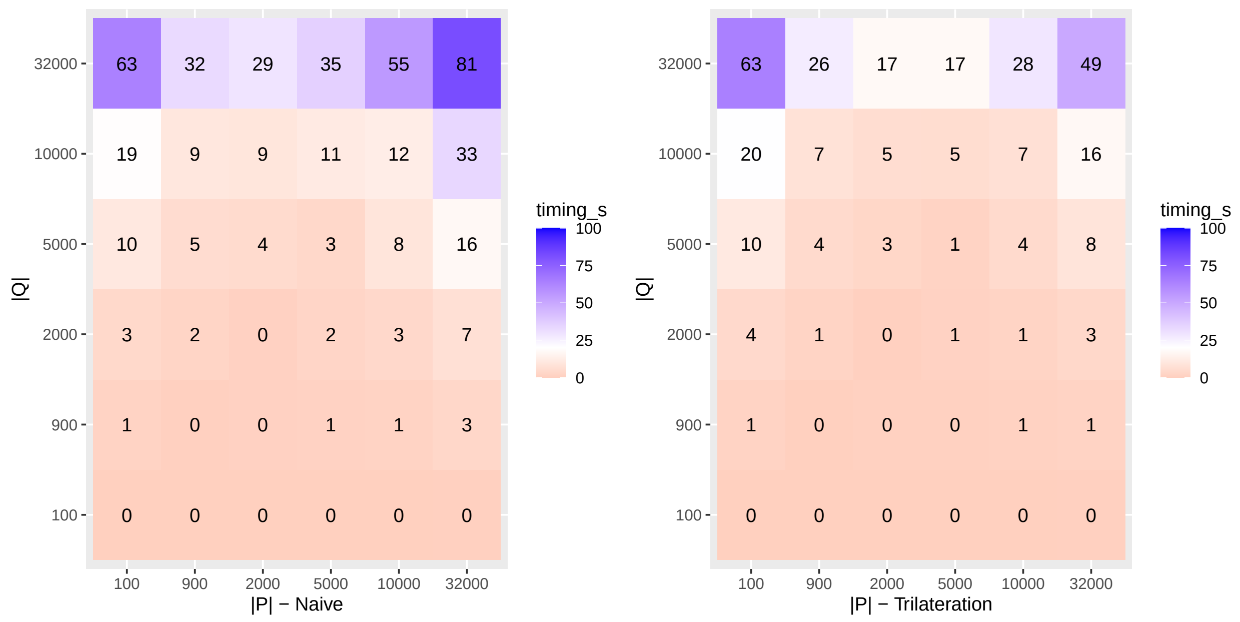
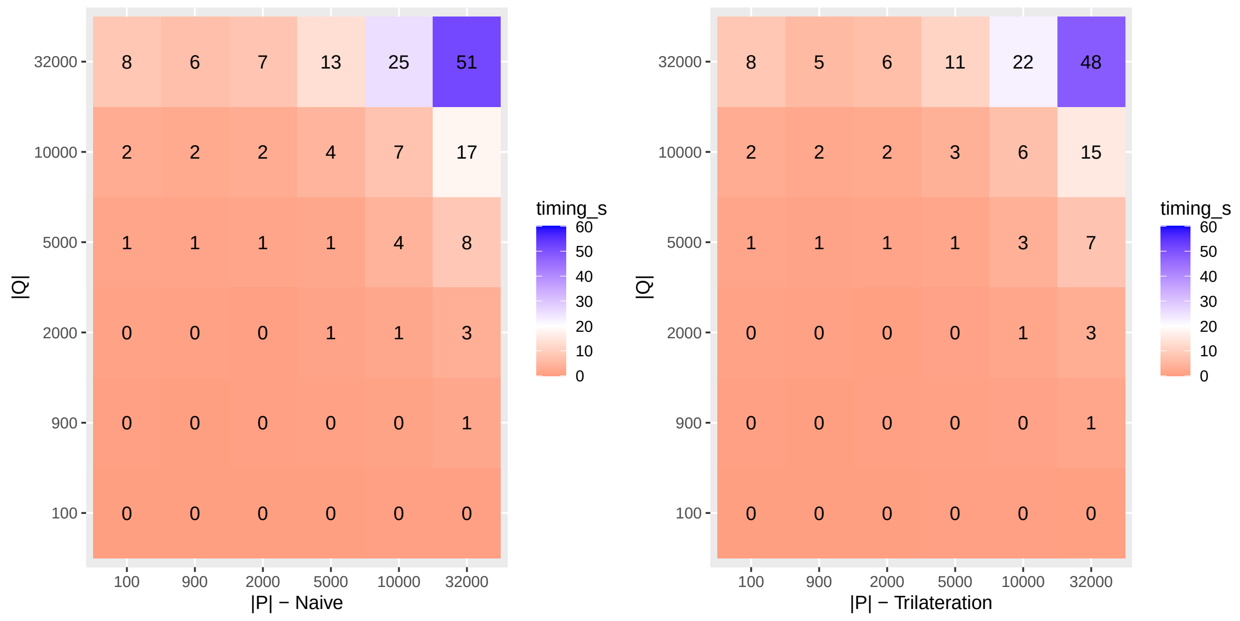


Figure 11: SQL Timings By Category (s) - d=10 - lower is better



|Q| \\ |P| − Naive	100	900	2000	5000	10000	32000
32000	8	6	7	13	25	51
10000	2	2	2	4	7	17
5000	1	1	1	1	4	8
2000	0	0	0	1	1	3
900	0	0	0	0	0	1
100	0	0	0	0	0	0

|Q| \\ |P| − Trilateration	100	900	2000	5000	10000	32000
32000	8	5	6	11	22	48
10000	2	2	2	3	6	15
5000	1	1	1	1	3	7
2000	0	0	0	0	1	3
900	0	0	0	0	0	1
100	0	0	0	0	0	0

Figure 12: SQL Timings By Category (s) - d=100 - lower is better

3.7.2.2 The diagonal is strangely fast.

This we originally thought of as a flaw in our experimental design, but now that we analyze it it's quite interesting and we chose not to change it. The situation along the diagonal is that we're using the same category for P and Q; this means that the datasets are identical, and that there is a 100% chance, always, of finding a point p within d of $q \forall q \in Q$ namely the point itself - since it is in the dataset and we made no effort to restrict this. There may or may not be a second point within distance, but recall that our definition only requires one (since we did not extend to $k - NA$). But why is this so fast? Given the guaranteed success of finding at least one qualifying point in every dataset, we surmise that the performance improvement is a statistical result of almost never having to iterate over the entire set P. The average time to find a point if one is guaranteed to exist, should be roughly half the time (over repeated iterations) of searching the whole set. The entire query time is then some statistical fraction of a prior expectation of the likelihood of finding a match.

Given our chart of d vs. NAP earlier, it makes sense that the diagonal's relative value is higher when d is low. That is, when $d = 100$ every point is adequate, even without itself as a guaranteed qualifying point, so there is no real performance gain – we never have to search every p for a match. When $d = 0.1$, on the other hand, we have to search the entire data set, in vain, over 50% of the time (per our chart), wasting an entire scan of P. Along the diagonal this never happens, and the time results mirror this fact.

3.7.2.3 The impact of varying distance d

Our best results are on low d with high $|Q|$ and high $|P|$. The NAIVE-NA query took nearly 15 minutes on our hardware to execute a 32,000x32,000 point search for network adequacy for $d = 0.1$. A little over 4 minutes for $d = 1$, and just over and under a minute for $d = 10$ and $d = 100$ respectively. A similar pattern exists for the $TRILAT - NA$ algorithm.

The reason for this seems straightforward... the smaller the distance d, the less likely it is that any given point p is within d of a query point q. Thus, in a large sample P we expect to have to search more points, and spend more time, as d decreases. This is a simple theory consistent with the result.

3.7.2.4 Complex relationship with |P| for NAIVE-NA

A more complicated relationship exists as we walk the charts from left to right. Examine the chart where $d = 1$ for a moment. On every but the bottom line (where zeroes probably hide this as a rounding problem), on the NAIVE-NA chart, the query time is HIGHER for $|P| = 100$ than it is for $|P| \in (900, 2000, 5000, 10000)$, however it picks UP again when $|P| = 32000$.

Why? We've run these tests repeatedly, in random order, back to back, to rule out hot/cold cache issues or other startup problems.

Let's look at the actual NAP values for that row:

Table 9: P vs. ~NAP for d=0.1, Q=10000

P	NAP
100	36.27
900	70.81
2000	76.37
5000	83.66
10000	100.00
32000	93.50

With only 100 candidate points, only 36% of Q is within 0.1 miles of a point in P. The other 64% must iterate all $|P| * |Q|$ comparisons (in the NAIVE-NA implementation), resulting in $0.64 * 10000 * 100 = 640,000$ comparisons plus a portion of the other 36% which is at most $0.36 * 10000 * 100 = 360,000$. Applying the same math for $|P| = 900$, at 70.8% (29.2 inadequate) we get $0.292 * 10000 * 900 = 2,628,000$. So. We see no direct theoretical observation that would make this make sense.

Sooo... we have to look at the SQL query plan:

```
Aggregate  (cost=37295851666.13..37295851666.14 rows=1 width=16)
     (actual time=86693.568..86693.570 rows=1 loops=1)
  ->  Nested Loop Left Join  (cost=0.00..37295851616.50 rows=9925 width=8)
         (actual time=152.251..86690.574 rows=10000 loops=1)
        ->  Seq Scan on sample_cat_ref_dists m
              (cost=0.00..4018.00 rows=9925 width=60)
              (actual time=151.943..166.146 rows=10000 loops=1)
               Filter: (category = 14)
               Rows Removed by Filter: 140000
        ->  Limit  (cost=0.00..3757768.00 rows=1 width=4)
                    (actual time=8.651..8.651 rows=0 loops=10000)
              ->  Seq Scan on sample_cat_ref_dists p
                    (cost=0.00..3757768.00 rows=1 width=4)
                    (actual time=8.649..8.649 rows=0 loops=10000)
                     Filter: category=10 ...
                     Rows Removed by Filter: 95609
Planning Time: 0.464 ms
JIT:
  Functions: 11
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 4.194 ms
  Inlining 9.301 ms
  Optimization 86.339 ms
  Emission 55.402 ms
  Total 155.236 ms
```

```
 Execution Time: 86697.917 ms
(15 rows)

Time: 86699.517 ms (01:26.700)
```

```
Aggregate  (cost=3107991549.46..3107991549.47 rows=1 width=16)
           (actual time=43705.837..43705.839 rows=1 loops=1)
   ->  Nested Loop Left Join  (cost=0.00..3107991499.83 rows=9925 width=8)
          (actual time=158.346..43702.035 rows=10000 loops=1)
         ->  Seq Scan on sample_cat_ref_dists m
               (cost=0.00..4018.00 rows=9925 width=60)
               (actual time=158.024..172.995 rows=10000 loops=1)
                Filter: (category = 14)
                Rows Removed by Filter: 140000
         ->  Limit  (cost=0.00..313147.33 rows=1 width=4)
                     (actual time=4.351..4.351 rows=1 loops=10000)
               ->  Seq Scan on sample_cat_ref_dists p
                    (cost=0.00..3757768.00 rows=12 width=4)
                    (actual time=4.349..4.349 rows=1 loops=10000)
                      Filter: category=11 ...
                      Rows Removed by Filter: 43992
 Planning Time: 0.189 ms
 JIT:
   Functions: 11
   Options: Inlining true, Optimization true, Expressions true, Deforming true
   Timing: Generation 1.702 ms
   Inlining 9.062 ms
   Optimization 91.460 ms
   Emission 56.565 ms
   Total 158.790 ms
 Execution Time: 43707.634 ms
(15 rows)

Time: 43708.659 ms (00:43.709)
```

There are two key differences here - the inner (second) “Rows Removed by Filter” which, for $|P| = 100$ is 95609 and for $|P| = 900$ is 43992. The other, related, difference being the actual time in the loops - 86693.568 for the outer and 8.649 for the inner when $|P| = 100$ and 43705.837 outer, 4.349 inner for $|P| = 900$. This indicates that our analysis was flawed; the SQL optimizer swapped what we presumed were the inner and outer loops, and given the percents, the inner loop is more expensive when $|P| = 100$ (by about double).

3.7.2.5 Complex relationship with |P| for TRILAT-NA

The pattern is similar, but not identical, in the right half of the graphs - those using our new TRILAT-NA algorithm.

If we examine the top rows for $d = 0.1$ and $d = 1$, where runtime is longest, the NAIVE-NA algorithm as we described may slightly increase performance for low $|P|$, but quickly sees a runtime increase as $|P|$ increases. The TRILAT-NA algorithm, however, while it isn’t monotonic for $|P| \in (100, 900)$, it actually *decreases* as $|P|$ increases from then on (with the exception of items on the diagonal, which are special as we discussed before).

This is easier to explain… the addition of the filters using the $refdist_i$ distances causes a much quicker elimination of points outside of the d radius from each q. More points in P makes the efficiency of these culls more pronounced. That is, if we look at the “Rows Removed by Filter” as we did in the explain before, and remember the figure [Monte Carlo Estimating Ring Overlap Area], we see that those three additional refdist filters will exclude a high percent of candidate records quickly. The *remaining* points, subject to the expensive st_distance calculation, have a high percent chance of being within d of q.

The higher the number of points in P, the more points culled as a result of the refdist filters that no longer have to have st_distance called, compared to the NAIVE-NA query, resulting in the performance gains we see.

3.7.2.6 NA Performance Conclusions

The TRILAT-NA algoritm was as fast or faster in almost every test we ran; the only major outlier was when $d = 0.1$ and $|P| = 100$, in two cases ($|Q| = 2000$ and $|Q| = 10000$), TRILAT-NA was slower by 2 and 14 seconds (about 7% and 10%) respectively. When $d = 1$, $|P| = 100$ again, and $|Q| = 900$, $|Q| = 5000$, or $|Q| = 10000$, TRILAT-NA was slower by 1 second or less in each case. Given that our resolution is 1-second, these are statistically minor. Even for the larger discrepancies, it seems that the TRILAT-NA algoritm is only slower for very small values of $|P|$, which is where optimization is least needed in the first place. When the workload takes the longest, the TRILAT-NA algoritm performs significantly faster than the naive algorithm.

3.8 Network Adequacy

The trilateration index was originally designed to improve efficiency of the "network adequacy" (NA) problem for health care. Network adequacy is a common legal requirement for medicare or insurance companies with constraints such as:

- 90% of members must live within 50 miles of a covered emergency room
- 80% of female members over the age of 13 must live within 25 miles of a covered OB/GYN
- 80% of members under the age of 16 must live within 25 miles of a covered pediatrician
- etc.

Note that these are all illustrative examples; the real "Medicare Advantage Network Adequacy Criteria Guidance" document for example, is a 75 page document.

Similar requirements, legal or otherwise, show up in cellular network and satellite communication technology (numbers are illustrative):

- Maximize the number of people living within 10 miles of a 5G cell tower
- 100% of all major highways should be within 5 miles of a 4G cell tower
- There must be at least 2 satellites within 200 km of a point 450 km directly above every ground station for satellite network connectivity at any given time
- There must be at least 1 satellite with access to a ground station within 50 km of a point 450 km directly above as many households as possible at any given time

The nearest-neighbor problem was called the "Post-Office Problem" in early incarnations, and the system of post offices lends itself to a similar construction: * Ensure that all US Postal addresses are within range of a post office

and so forth.

3.8.1 Formalization of Network Adequacy

We formalize the concept of "Network Adequacy" as:

3.8.1.1 Network Adequacy Definition

Given a non-empty set of points P and a non-empty set of query points Q in a metric space M (where $P \cap Q$ comprises the 'network'), the network is 'completely adequate' for a distance d and a distance function $D(a, b)$ describing the distance between points a and b for $a \in M$ and $b \in M$ if for every point q (where $q \in Q$) $\exists$ at least one point p $(p \in P) \ni D(p, q) <= d$. Otherwise the network is 'inadequate'.

We call a single point q 'adequate' itself, if it satisfies the same condition – i.e. $\exists$ at least one point p $(p \in P)$ $D(p, q) <= d$.

3.8.1.2 Network Adequacy Percent

If, within P, we consider the largest subset $P' \in P$ where P' is 'completely adequate', then P has a "Network Adequacy Percent (NAP)" of $|P'|/|P|$. Note that P' can be defined (identically) as the union of all 'adequate' points $p \in P$.

3.8.1.3 Network Adequacy Threshold

We can generalize this slightly more by describing a network as 'adequate with threshold T' by introducing a percent T $(0 <= T <= 1)$ such that the same network is adequate if at least $T * |Q|$ (or T percent of points in Q) are individually 'adequate'.

Another way of saying this is that the netwrk is 'adequate with threshold T' if the Network Adequacy Percent $NAP > T$.

In this case, if $T == 1$ we have the original case. If $T == 0$ we have a trivial case where the network is always adequate (even if Q and/or P are empty, which is generally disallowed).

3.8.1.4 k-Network Adequacy (kNA)

Similarly to the kNN extension of nearest-neighbor search, where we want the k nearest values, a k-network adequacy problem could be stated:

Given P and Q as before, a network is 'completely k-adequate' for a given k if $\exists$ at least k points $p \in P \ni D(p, q) <= d$. Otherwise the network is 'k-inadequate'.

Similarly, a single point q is 'k-adequate' itself, if it satisfies the same condition – i.e. $\exists$ at least k points $p \in P \ni D(p, q) <= d$. Otherwise the point is 'k-inadequate'.

3.8.1.5 Solving NA Using NN Algorithms

Generally, using NN techniques, the brute-force is iterative: for each point q, find the nearest point p and if $D(p, q) < d$ count it as conforming, otherwise count it as non-conforming. We then calculate the ratio of $r = \frac{conforming}{|Q|}$ to determing whether $r >= T$ or not.

If we set $m = |Q|$ and $n = |P|$, and if we use a Nearest-Neighbor algorithm with $O(n\ log\ n)$ then the time complexity for Network Adequacy becomes $O(m * n\ log\ n)$.

In the worst-case, we cannot improve on this, however we can introduce what we believe are novel algorithms, based on the Trilateration Index, which execute efficiently compared to this iterative approach, by deeply reducing the search space and number of times the distance functions must be called in typical real-world cases. Also, trivially, it is unnecessary to find the nearest point p, merely prove that at least one such point exists (or none exists) for a given q.

3.9 Multilateration NA Algorithms

In SQL, we implement two NA algorithms to compare this theoretical setup to a typical real-world example detecting whether there exists a record in P within d of each of a set of query points Q:

- "NAIVE-NA" - the default SQL Query algorithm
- "TRILAT-NA" - the approach we've described

3.9.1 NAIVE-NA

The most basic SQL query, in a database that has Geospatial extensions, to calculate Network Adequacy is something like:

```sql
    select count(q.sampleid) as qcount,
        count(p2.sampleid) as tcount
    from q_points q
    left join lateral (select p.sampleid from p_points p
        where
          st_distance(p.st_geompoint, q.st_geompoint) <= (1609.34 * mydist)
        limit 1
    ) p2 on true
```

This assumes two tables - "q_points" and "p_points" containing the points in Q and P respectively. Each has a field "st_geompoint" containing a geospatial position for each point. The "st_distance" function is a SQL function to calculate the distance between two points - in our case we need to ensure the database uses the accurate Geodesic calculation from our research.(Karney 2013)

This returns the number of records in Q as qcount and the number of records in P as tcount. The Network Adequacy Percent is then $\frac{tcount}{qcount}$.

Note that this function is in PostgreSQL syntax; it requires slight moderation but otherwise works (we tested) in Microsoft SQL Server and MySQL. It likely works with little modification in any database which supports the SQL:1999 standard for lateral joins and geospatial points and distance functions. One thing that is NOT identical between database implementations is the ability or effectiveness of database indexes on this query. In postgres, we have experimented and found that

Also note the 1609.34 - this is to convert the distance from meters to miles, which is not core to the algorithm, but left here since these are the units we work with in our experimental results, and as an example.

3.9.2 TRILAT-NA

Recall that we require fixed reference points for Trilateration, and per our previous construction, we selected these:

- Point 1: $90.000000, 0.000000$ (The geographic north pole)
- Point 2: $38.260000, -85.760000$ (Louisville, KY on the Ohio River)
- Point 3: $-19.22000, 159.93000$ (Sandy Island, New Caledonia)

The "NAIVE-NA" query requires no real setup, other than storing the data from the P and Q datasets. Not so here – we require additional fields added to the database in the q_points and p_points tables to store the distances from each point to these reference points. We name those fields $refdist1$, $refdist2$, and $refdist3$. Recall that this is a one-time setup requiring $3 * |P| + 3 * |Q|$ calls to st_distance.

The SQL implementation of the TRILAT-NA algorithm then looks like this:

```
select count(q.sampleid) as qcount,
    count(p2.sampleid) as tcount
from q_points q
left join lateral (select p.sampleid from p_points p
    where
      abs(q.refdist1 - p.refdist1) <= (1609.34 * mydist)
      and abs(q.refdist2 - p.refdist2) <= (1609.34 * mydist)
      and abs(q.refdist3 - p.refdist3) <= (1609.34 * mydist)
      and st_distance(p.st_geompoint, q.st_geompoint) <= (1609.34 * mydist)
    limit 1
) p2 on true
```

Note that this is identical to the query for NAIVE-NA, with the addition of the three lines comparing the refdist values.

These accomplish two things:

1. They allow SQL to optimize using normal (non-geospatial) database indexes when comparing between the two points p and q, using simple subtraction rather than a complicated geodesic query.

2. They allow for three opportunities to reduce the dataset size before the high cost geodesic query is performed. Recall our figure [Monte Carlo Estimating Ring Overlap Area] that exhibited how the area of the three overlapping rings of width d was << the area of the search space; a similar thing happens here… For a given distance d, the set of points where $q.refdist_i - p.refdist_i <= d$, is the intersection of those three rings of width d with centers on the three reference points and with diameters such that

the middle of each ring passes through q. This eliminates most points if d is relatively small - small enough that some points q are inadequate is generally a good test.

See the Experimentation and Experimental Results sections for details on our specific implementations, tests, and results.

3.10 Multilateration NN Algorithms

We create a total of four similar algorithms to exploit the Multilateration Index, in particular for geodesic distances (although our implementations are applicable to any distance function): Trilateration (TRI), TrilaterationApprox (TIA), TrilaterationExpand (TIE), and TrilaterationExpand2 (TIE2).

3.10.1 Comparing Algorithms

Recall from our Review of Literature, we are comparing NN algorithms by four areas Training Time Complexity, Memory Space, Prediction Time Complexity, Insertion/Move Complexity.

For each of these Trilateration algorithms, complexity is:

1. Training Time Complexity is $O(|P|)$ - each point $p \in P$ is compared to the $d+1$ reference points ($d+1=3$ for Trilateration).
2. Memory Space is $|P| * (d+1)$ since each structure requires the storage of a $d * p$ array of $d+1$ distances for each point $p \in P$
3. Prediction Time Complexity is, as with other kNN algorithms, bounded by worst-case of $O(n)$. The algorithms may differ in their average case performance.
4. Insertion Complexity is $O(|P|)$ - being the cost if inserting or updating an element in the sorted array of distances for each $p \in P$.

3.10.1.1 Trilateration (TRI)

The main trilateration algorithm for exact Nearest-Neighbor solutions takes the Trilateration Index (recall - the distances stored in a sorted array form from all points in P with respect to $d+1$ fixed reference points – 3 in the case of 2-d Trilateration) and applies the following for a query point q. This provides simple a simple $O(log(n))$ binary search by distance, along with the ability to quickly iterate through points consecutively closer or farther from a given point in list order using common array operations.(Tainiter 1963)

```
Calculate qd1..qdn as the distances from point q to the n reference points r1..rn
Find the index i1 in TI for the nearest point along the d1 distance to q
Create HEAP - a max heap of size k
    Let WORST_DIST bet the maximum distance on HEAP at any time
Calculate LOW_IDX = i1-(k/2) and HIGH_IDX = i1+(k/2)
For all points c in TI between TI[LOW_IDX] and TI[HIGH_IDX]:
    push c onto HEAP
Find the index LOW_IDX_POSSIBLE in TI as:
    the highest point along d1 where |TI[,d1]-qd1| > WORST_DIST
Find the index HIGH_IDX_POSSIBLE in TI as:
    the lowest point along d1 where |TI[,d1]-qd1| > WORST_DIST
While LOW_IDX > LOW_IDX_POSSIBLE or HIGH_IDX < HIGH_IDX_POSSIBLE:
    Choose the closer of TI[LOW_IDX-1, d1] or TI[HIGH_IDX+1, d1] along d1 (call it c)
        If ALL of |TI[c, dx]-qx| (for all x 2..n) are < WORST_DIST
          Calculate D(q, c)
          If D(q, c) < WORST_DIST:
              Add c to the HEAP
              recalculate LOW_IDX_POSSIBLE and HIGH_IDX_POSSIBLE
        depending on the choice, decrement LOW_IDX or increment HIGH_IDX
Return HEAP
```

Basically, looking only along one distance dimension (proximity to d1), find the closest k points to q (which is very quick along a sorted array - O(log n) to find the first point and O(1) to add and subtract k/2 to the indices to get the boundary). Expand the low and high values (selecting the next point as the closest of the 2 along d1) until we have k points such that the farthest (worst) distance to one of those points is closer than the distance along d1 for any other point (which is bounded by LOW_IDX_POSSIBLE and HIGH_IDX_POSSIBLE, since that distance is our sort order).

3.10.1.2 TrilaterationApprox (TIA)

In an attempt to gain benefit from the relaxed constraints of an "approximate" aNN approach, we experiment with an algorithm that effectively excludes the distance calculations altogether, from our TRI algorithm altogether. Recall that, for a given d_x distance to reference point r_x, a point p can be *no closer than* $q_{dx} - p_{dx}$; if we treat the approx_distance $(q, p) = \frac{\sum_{x=1}^{m} q_{dx} - p_{dx}}{|d|}$ (the mean of the relative distances from q to all reference points r_x), or, more aggressively, the minimum such distance, we can return an approximate result without ever having to call the distance function itself (after the index is created).

We ended up abandoning this approach after only a few tests - there was a significant drop in recall (the mechanism by which ann-benchmarks measures effectiveness of aNN algorithms), with no particular improvement in performance. This effectively removed aNN from our consideration; our results are primarily focused on exact NN results as a consequence.

3.10.1.3 TrilaterationExpand (TIE)

We theorized that the TRI approach may be slowed down by the overhead of having to iterate one point at a time, and by not utilizing more than one reference point early in the process. Given the efficiency of the $Within()$ function (see Time Complexity), we wondered if we should treat the distance d as the target variable, and use an incremental search to zero in on the proper value to result in k neighbors within d.

This turns out to be a silly idea, once we get the results back, but here we are. The algorithm would look like:

```
set radius = 0.5
set too_low = 0
set too_high = maximum possible distance in the space
set x = CountWithin(radius, q, P)
while x != k:
    if x < k:
        set too_low = radius
        set radius = (radius + too_high)/2
    else:
        set too_high = radius
        set radius = (radius + too_low)/2
return Within(radius, q, P)
```

3.10.1.4 TrilaterationExpand2 (TIE2)

Another approach to minimizing the overhead of expanding the range in the TRI algorithm by one at a time is to simply expand by some fixed amount > 1. This actually shows performance gains when the distance functions are inexpensive, although not enough to really be competitive with other NN solutions, but shows no benefit (in fact, it incurs quite the cost) when using our expensive geodesic distance functions. See our results section for more details.

Fundamentally, the change to the TRI algoritm is that we expand by k (which is a convenient constant), or some larger constant, at a time, rather than 1 point along d_1 in TI. In effect:

```
Set CHUNK equal to the greater of k or 500
Calculate qd1..qdn as the distances from point q to the n reference points r1..rn
Find the index i1 in TI for the nearest point along the d1 distance to q
Create HEAP - a max heap of size k
    Let WORST_DIST bet the maximum distance on HEAP at any time
```

```
Calculate LOW_IDX = i1-(k/2) and HIGH_IDX = i1+(k/2)
For all points c in TI between TI[LOW_IDX] and TI[HIGH_IDX]:
    push c onto HEAP
Find the index LOW_IDX_POSSIBLE in TI as:
    the highest point along d1 where |TI[,d1]-qd1| > WORST_DIST
Find the index HIGH_IDX_POSSIBLE in TI as:
    the lowest point along d1 where |TI[,d1]-qd1| > WORST_DIST
While LOW_IDX > LOW_IDX_POSSIBLE or HIGH_IDX < HIGH_IDX_POSSIBLE:
    If TI[LOW_IDX-1, d1] is closer than TI[HIGH_IDX+1, d1]:
        PRIOR_IDX = LOW_IDX
        LOW_IDX = LOW_IDX - CHUNK
        Evaluate points between TI[LOW_IDX,] and TI[PRIOR_IDX,]:
          If ALL of |TI[c, dx]-qx| (for all x 2..n) are < WORST_DIST
          Calculate D(q, c)
          If D(q, c) < WORST_DIST:
              Add c to the HEAP
    else:
        PRIOR_IDX = HIGH_IDX
        HIGH_IDX = HIGH_IDX + CHUNK
        Evaluate points c between TI[PRIOR_IDX,] and TI[HIGH_IDX,]:
          If ALL of |TI[c, dx]-qx| (for all x 2..n) are < WORST_DIST
          Calculate D(q, c)
          If D(q, c) < WORST_DIST:
              Add c to the HEAP
    recalculate LOW_IDX_POSSIBLE and HIGH_IDX_POSSIBLE
Return HEAP
```

3.10.1.5 Implementation Notes

There's a lot to digest in these algorithms; many choices were made, and various performance issues were encountered. Of note, many of the implementation specifics were due to building our algorithms on top of the existing scikit-learn code.(Pedregosa et al. 2011) The use of the $HEAP$ structure, mentioned in our code, is immediately attributable to leveraging scikit-learn's source. Also, being built in Cython, and having been field-tested for about 10 years, it's possible our code could be improved or may have bugs compared to the rest of scikit-learn, but we tried real hard.

As mentioned before, the TIA algorithm failed dominantly because either performance or recall were too slow… recall was a problem when only 1 reference point was used, and performance faltered with multiple reference points due to the high cost of calculating the intersection of candidate point lists from multiple reference points.

The implementation on both "Expanding" algorithms (TIE and TIE2) presented many choices. Initially we sought to find a reasonable initial guess for the candidate *radius*, however no suitable algorithm presented itslef that was superior to guessing "0.5" as a first guess (curiously true regardless of the coordinate scale). Similarly, the value of 500 for $CHUNK$ size was found reasonable via trial and error, although heuristics to arrive at the number, rather than hard-coding it, could probably benefit specific cases.

See Experimental Results and Experimentation for the results and specifics of how we tested these algorithms.

---

4 Appendixy stuff

4.1 Code and datasets

All code including markdown and custom sample data files to generate this document are available here: https://github.com/chipmonkey/TrilaterationIndex

A custom fork and branch of scikit-learn which includes the Python/Cython implementation of our Multilateration Nearest-Neighbor and related distance functions is available here: https://github.com/chipmonkey/scikit-learn/tree/feature/chip-test-index

A custom fork and branch of the ANN Benchmarks code which includes hooks to our Multilateration NN implementation (from the custom scikit-learn fork) is available here: https://github.com/chipmonkey/ann-benchmarks

Bibliography

Abel, Jonathan S., and James W. Chaffee. 1991. "Existence and Uniqueness of Gps Solutions." *IEEE Transactions on Aerospace and Electronic Systems* 27 (6). https://doi.org/10.1109/7.104271.

Ahmadi, Seyed Alireza, Vahid Vahidinasab, Mohammad Sadegh Ghazizadeh, Kamyar Mehran, Damian Giaouris, and Phil Taylor. 2019. "Co-Optimising Distribution Network Adequacy and Security by Simultaneous Utilisation of Network Reconfiguration and Distributed Energy Resources." *IET Generation, Transmission and Distribution* 13 (20). https://doi.org/10.1049/iet-gtd.2019.0824.

ASPRS. 2015. "ASPRS Positional Accuracy Standards for Digital Geospatial Data." *Photogrammetric Engineering & Remote Sensing* 81 (3). American Society for Photogrammetry; Remote Sensing: 1–26. https://doi.org/10.14358/pers.81.3.a1-a26.

Aumüller, Martin, Erik Bernhardsson, and Alexander Faithfull. 2020. "ANN-Benchmarks: A Benchmarking Tool for Approximate Nearest Neighbor Algorithms." *Information Systems* 87: 101374. https://doi.org/https://doi.org/10.1016/j.is.2019.02.006.

Bentley, Jon Louis. 1975. "Multidimensional Binary Search Trees Used for Associative Searching." *Communications of the ACM* 18 (9). https://doi.org/10.1145/361002.361007.

Brian Beck, et. al. n.d. *Geopy.* https://github.com/geopy/geopy.

Brummelen, Glen Van. 2012. *Heavenly Mathematics: The Forgotten Art of Spherical Trigonometry. Heavenly Mathematics: The Forgotten Art of Spherical Trigonometry.* https://doi.org/10.33137/aestimatio.v11i0.26065.

Chen, George H., and Devavrat Shah. 2018. "Explaining the Success of Nearest Neighbor Methods in Prediction." *Foundations and Trends in Machine Learning* 10 (5-6). https://doi.org/10.1561/2200000064.

Cohen, Hagai, and Ely Porat. 2010. "Fast Set Intersection and Two-Patterns Matching." In *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics).* Vol. 6034 LNCS. https://doi.org/10.1007/978-3-642-12200-2_22.

Dong, Wei, Moses Charikar, and Kai Li. 2011. "Efficient K-Nearest Neighbor Graph Construction for Generic Similarity Measures." In *Proceedings of the 20th International Conference on World Wide Web, WWW 2011.* https://doi.org/10.1145/1963405.1963487.

Fu, Cong, Chao Xiang, Changxu Wang, and Deng Cai. 2018. "Fast Approximate Nearest Neighbor Search with the Navigating Spreading-Out Graph." In *Proceedings of the VLDB Endowment.* Vol. 12. https://doi.org/10.14778/3303753.3303754.

Gade, Kenneth. 2010. "A Non-Singular Horizontal Position Representation." *Journal of Navigation* 63 (3). Cambridge University Press: 395–417. https://doi.org/10.1017/S0373463309990415.

Indyk, Piotr, and Rajeev Motwani. 1998. "Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality." In *Conference Proceedings of the Annual ACM Symposium on Theory of Computing*. https://doi.org/10.4086/toc.2012.v008a014.

Johnson, Jeff, Matthijs Douze, and Hervé Jégou. 2017. "Billion-Scale Similarity Search with Gpus." *arXiv Preprint arXiv:1702.08734*.

Kaplan, Philip. 2018. "Ancient Geography: The Discovery of the World in Classical Greece and Rome." *The AAG Review of Books* 6 (1). https://doi.org/10.1080/2325548x.2018.1402263.

Karney, Charles F.F. 2013. "Algorithms for Geodesics." *Journal of Geodesy* 87 (1). https://doi.org/10.1007/s00190-012-0578-z.

Kazel, Sidney. 1972. "MULTILATERATION Radar." *Proceedings of the IEEE* 60 (10). https://doi.org/10.1109/PROC.1972.8886.

Lambert, W. D. 1942. "The Distance Between Two Widely Separated Points on the Surface of the Earth." *J. Washington Academy of Sciences*, no. 32 (5).

Lee, Harry B. 1975. "A Novel Procedure for Assessing the Accuracy of Hyperbolic Multilateration Systems." *IEEE Transactions on Aerospace and Electronic Systems* AES-11 (1). https://doi.org/10.1109/TAES.1975.308023.

Linares, Jean Marc, Santiago Arroyave-Tobon, José Pires, and Jean Michel Sprauel. 2020. "Effects of Number of Digits in Large-Scale Multilateration." *Precision Engineering* 64: 1–6. https://doi.org/https://doi.org/10.1016/j.precisioneng.2020.03.009.

Liu, Ting, Andrew W. Moore, and Alexander Gray. 2006. "New Algorithms for Efficient High-Dimensional Nonparametric Classification." *Journal of Machine Learning Research* 7. https://doi.org/10.7551/mitpress/4908.003.0008.

Mahdavi, Meisam, and Elham Mahdavi. 2011. "Transmission Expansion Planning Considering Network Adequacy and Investment Cost Limitation Using Genetic Algorithm." *World Academy of Science, Engineering and Technology* 80. https://doi.org/10.5281/zenodo.1086125.

Paulevé, Loïc, Hervé Jégou, and Laurent Amsaleg. 2010. "Locality Sensitive Hashing: A Comparison of Hash Function Types and Querying Mechanisms." *Pattern Recognition Letters* 31 (11). https://doi.org/10.1016/j.patrec.2010.04.004.

Pedregosa, F., G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, et al. 2011. "Scikit-Learn: Machine Learning in Python." *Journal of Machine Learning Research* 12: 2825–30.

Pennington, Jeffrey, Richard Socher, and Christopher D. Manning. 2014. "GloVe: Global Vectors for Word Representation." In *Empirical Methods in Natural Language Processing (Emnlp)*, 1532–43. http://www.aclweb.org/anthology/D14-1162.

Prokhorenkova, Liudmila. 2019. "Graph-Based Nearest Neighbor Search: From Practice to Theory." *arXiv*.

Singh, Santar Pal, and S. C. Sharma. 2015. "Range Free Localization Techniques in Wireless Sensor Networks: A Review." In *Procedia Computer Science*, 57:7–16. Elsevier. https://doi.org/10.1016/j.procs.2015.07.357.

Strohmeier, Martin, Ivan Martinovic, and Vincent Lenders. 2018. "A K-Nn-Based Localization Approach for Crowdsourced Air Traffic Communication Networks." *IEEE Transactions on Aerospace and Electronic Systems* 54 (3). Institute of Electrical; Electronics Engineers Inc.: 1519–29. https://doi.org/10.1109/TAES.2018.2797760.

Tainiter, M. 1963. "Addressing for Random-Access Storage with Multiple Bucket Capacities." *Journal of the ACM (JACM)* 10 (3). https://doi.org/10.1145/321172.321178.

Tillquist, Richard C., and Manuel E. Lladser. 2016. "Metric-Space Positioning Systems (Mps) for Machine Learning." In *Proceedings of the 7th Acm International Conference on Bioinformatics, Computational*

Biology, and Health Informatics, 479. BCB '16. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/2975167.2985641.

Vincenty, T. 1975. "DIRECT and Inverse Solutions of Geodesics on the Ellipsoid with Application of Nested Equations." *Survey Review* 23 (176). Taylor & Francis: 88–93. https://doi.org/10.1179/sre.1975.23.176.88.

Wasserstein, Ronald L., Malvin H. Kalos, and Paula A. Whitlock. 1989. "Monte Carlo Methods, Volume 1: Basics." *Technometrics* 31 (2). https://doi.org/10.2307/1268841.

Wishner, Jane B, and Jeremy Marks. 2017. "Ensuring Compliance with Network Adequacy Standards: Lessons from Four States." *Robert Wood Johnson Foundation*, no. March.

Zhang, Yipeng, Fan Zhang, Yang Wang, Yulin Ma, and Honghai Li. 2017. "Localization of Nearest-Neighbour and Multilateration Analyse Based on Rfid." In *Proceedings - 2nd International Conference on Smart City and Systems Engineering, ICSCSE 2017*. https://doi.org/10.1109/ICSCSE.2017.64.

Zhou, Junyi, and Jing Shi. 2009. "RFID Localization Algorithms and Applications-a Review." *Journal of Intelligent Manufacturing* 20 (6). https://doi.org/10.1007/s10845-008-0158-5.