Detect Sparse Observation Zones
Senior Design Group May14-31

Final Report

Team Members:
John Harding
Nicholas McLaren
Michael Ore
Andrew Upah
Bryce Wilson

Advisor:
Dr. Ruchi Chaudhary, Department of Biology, University of Florida

Client:
Prof. Gordon Burleigh, Department of Biology, University of Florida

Problem Statement

There are a number of big databases that share biodiversity data for biological research and collaboration. For example, the Global Biodiversity Information Facility (http://www.gbif.org) has millions of species records such as latitude and longitude coordinates. In addition, the Avian Knowledge network (http://www.avianknowledge.net) has over 100 million bird observation records. These databases provide an enormous amount of information that is useful in understanding the patterns and dynamics of various species across a region. Another interesting, yet not well explored, direction is finding out the biggest regions where there are few or no species records. This may represent different features in the landscape or areas where there has been little or no sampling. This project is focused on developing a software tool that inputs a collection of millions of points and outputs the largest areas where there are few or no observations. More formally, this problem is written in the following way:

Problem (Detect Sparse Observation Zones):

Input: a collection of latitude and longitude points: k, n.

Output: n latitude and longitude points with corresponding radii such that there are no more than k points in the respective circle of each point.

Term and Acronym Definitions

CGAL: Open source software library that provides efficient algorithms in computational geometry
Convex Hull: the smallest convex region that contains any set of items
Qt: Cross-platform application framework used in creating graphical user interfaces
Input point: One of the coordinates in the input data set
(ordinary) Voronoi Diagram: A division of space using a set of input points in which each point has the corresponding region of all points in space nearer to that point than any other

Higher Order Voronoi Diagram: A generalization of Voronoi Diagrams. For any natural number k and given a set of input points, an order-n Voronoi Diagram associates every possible combination of k input points with the region where those n points are closest
nth Order Voronoi Diagram: A Higher Order Voronoi Diagram with a specific value of n
nth Degree Voronoi Diagram: Another generalization, similar to Higher Order Voronoi Diagrams. Every region is associated with a single input point that is the nth-closest input point

relative to every point in the region. We're generally interested in (k+1)th Degree Voronoi Diagrams

k-d tree: Multi-dimensional generalization of a binary search tree (Note: k is number of spatial dimensions for the tree, not related to the k parameter for the algorithm)

k-circle: Relative to a set of input points, a k-circle is a circle with fewer than k points inside it

n-distance function: A function — shorthand $d_n(x, y)$ — that gives the distance from the point (x, y) to the n-th closest data point. Generally, we're interested in the (k+1)-distance function — $d_{k+1}(x, y)$

search space: Region in which we are interested in finding k-circles. The centers of k-circles should be inside the search space, the boundary of the circle may extend outside.

local maximum: For a function f(x, y), a local maximum is a specific point (x_0, y_0) for which $f(x_0, y_0) \geq f(x, y)$ for all points (x, y) that are very close to (x_0, y_0)

global maximum: For a function f(x, y), a global maximum is a specific point (x_0, y_0) for which $f(x_0, y_0) \geq f(x, y)$ for all points (x, y) in the search space ("global maximum" may also refer to the corresponding value $f(x_0, y_0)$)

gradient: The gradient for a function f(x, y) at a specific point (x_0, y_0) is a vector whose direction is where the function increases most steeply and whose magnitude is the slope of the function in that direction

sampling point: A point on a regular grid covering the search space, where a function is evaluated

nearest-neighbor interpolation: A method of approximating a function's value between sampling points using the value of the closest sampling point

Design Requirements

Functional

1. Algorithm must take as input a list of coordinates and a number k
2. Algorithm must output circles containing fewer than k points
3. GUI must allow inputting parameters and running the algorithm

Non-Functional

1. Algorithm should output large circles, near optimal
2. Algorithm must run in a reasonable time for large inputs
3. Application must be able to run on Windows/Linux/MacOS
4. GUI should be intuitive and aesthetically pleasing

Project Design

Overview

The algorithm we delivered:

1. Create a regularly spaced grid of points (samples) over some search space
2. At each sample point, find the largest radius that contains at most k input points
3. Output the samples and corresponding radii as k-circles

This effectively measures how sparse the data is across the whole search space. The sampled k-circles with largest radii are nearly optimal.

Algorithm Design Approach

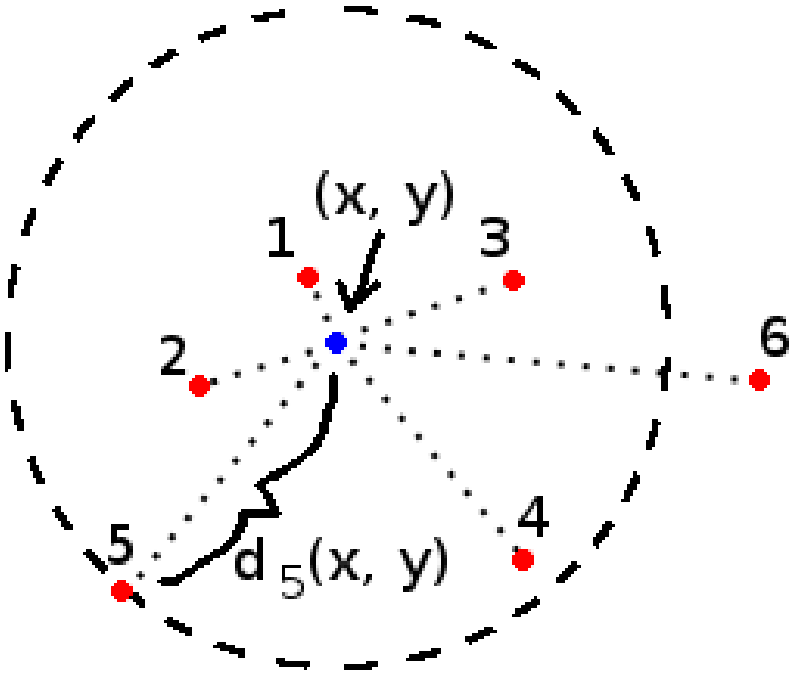

Fig. 1: Radius of $d_{k+1}(x, y)$ for k = 4

We start by defining a function that measures how sparse the input points are around any point in the search space. We define the (k+1)-distance function ($d_{k+1}(x, y)$) as the distance from the point (x, y) to the (k+1)th-closest input point. This is the largest possible radius for a k-circle centered at (x, y), as the circle would contain k+1 input points if the radius were expanded any further. The problem of finding large k-circles is the same as the problem of finding values of (x, y) for which $d_{k+1}(x, y)$ is large, in other words optimizing $d_{k+1}(x, y)$.

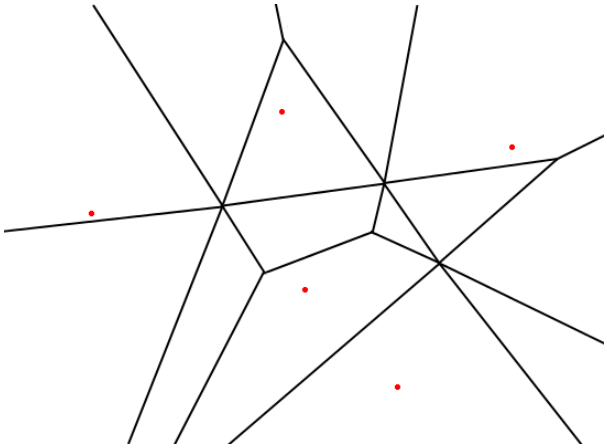

Fig. 2a: 2nd degree Voronoi diagram

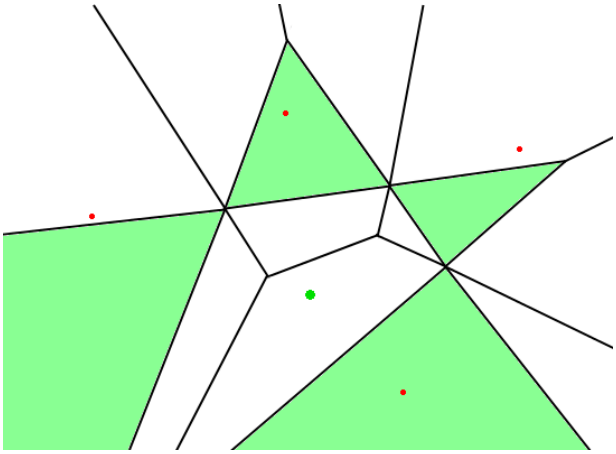

Fig. 2b: Green region has green point as 2nd closest

$d_{k+1}(x, y)$ is a piecewise function, where each "piece" is a region in the (k+1)th degree Voronoi diagram of the input points. The value of $d_{k+1}(x, y)$ within each region is the distance to the input point corresponding to that region. See **Fig. 2b**, where $d_{k+1}(x, y)$ for (x, y) in the green region is equal to the distance from (x, y) to the green point.

The surface (3D graph) of $d_{k+1}(x, y)$ within each region consists of a portion of an upside-down cone centered at the corresponding input point (which may be outside the region). On region edges, multiple input points are tied for (k+1)th closest and so the function values for both regions matchup and the function is continuous. The gradient of $d_{k+1}(x, y)$ is a unit vector pointing away from the (k+1)th closest input point. Because the magnitude of the gradient is bounded, we know that $d_{k+1}(x, y)$ changes slowly over distance.

We can compute $d_{k+1}(x, y)$ with a k-d tree. With the input points stored in a k-d tree, the (k+1)th closest input point from any point (x, y) can be found in O(k log N), where N is the total number of input points.

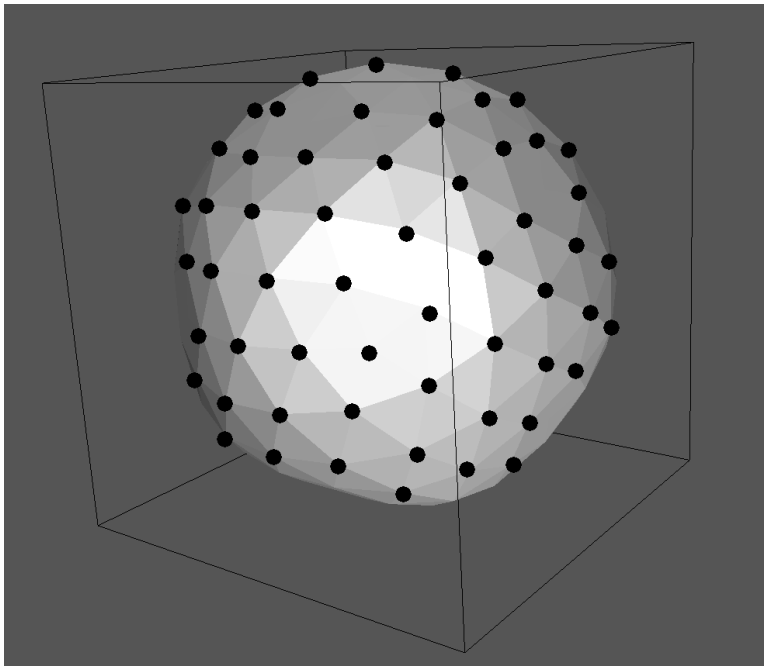

Fig. 3: Geodesic grid, sampling at vertices

For the most generality, we assume that we're dealing with a global data set and the entire Earth is the search space. Because of issues with wrap-around on planar maps, we model the Earth as a spherical globe and put the input points on its surface. To understand how $d_{k+1}(x, y)$ behaves over the search space, we cover it with a regular grid of sampling points and evaluate $d_{k+1}(x, y)$ at each one.

There are multiple ways to generate a grid of points on a sphere, and none are perfect. We decided on using the vertices of a geodesic grid (**Fig. 3**); a polyhedron that approximates a sphere. These vertices are not perfectly evenly spaced, but they are close and the exact spacing around any vertex can easily be calculated. Geodesic grids are created by iteratively subdividing a simpler polyhedron, and so the grid spacing is controlled by specifying the number of iterations. We use an icosahedron (20-sided basic solid) as a base and use Loop subdivision to refine it.

After generating this grid, $d_{k+1}(x, y)$ is sampled at each vertex. These samples provide a good idea for what $d_{k+1}(x, y)$ looks like over the whole search space and an approximation to the global maximum. Our code doesn't currently consider the possibility of a search space smaller than the entire surface of the Earth, but if a user is interested in a smaller area then they can focus their attention on the sampling points in their area of interest.

Error Analysis

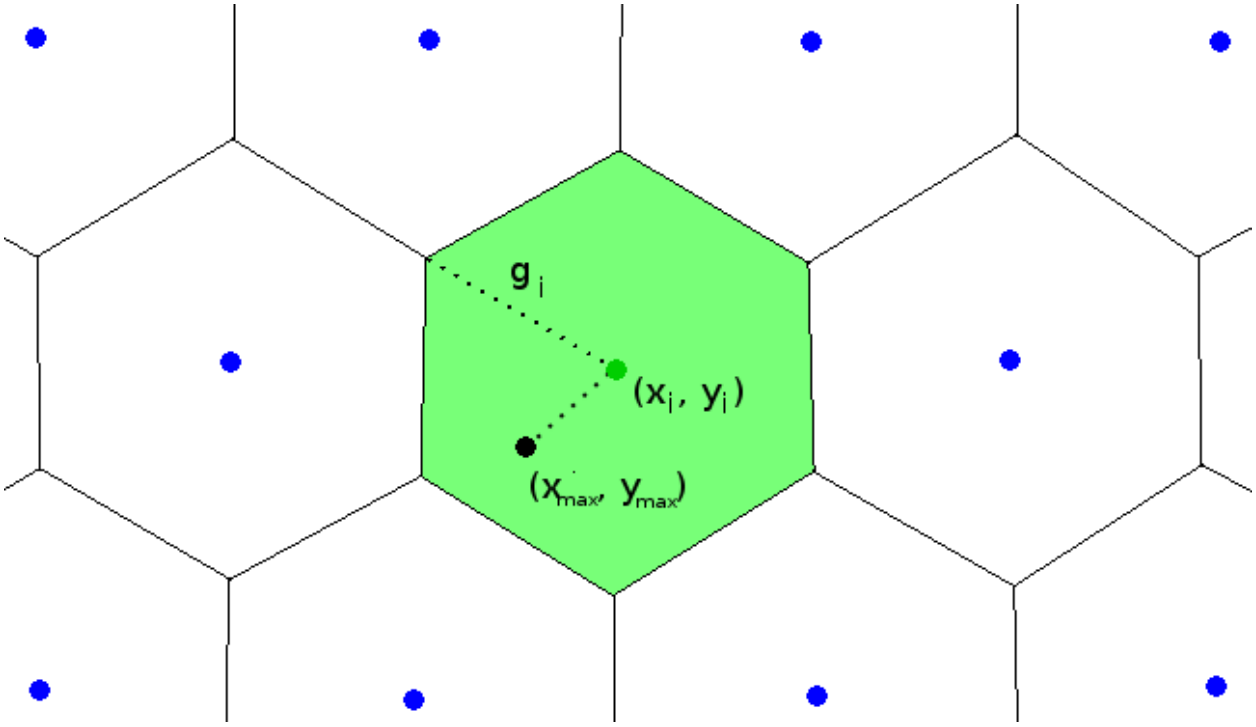

Fig. 4: Voronoi diagram of sampling points, g_i = grid spacing

The space between sampling points can be interpolated with nearest-neighbor interpolation. This means that, at any point, $d_{k+!}(x, y)$ can be approximated with the value of the sampling point closest to (x, y). Because of the way the sampling points are laid out, the Voronoi diagram consists of hexagonal cells for each point. For each sampling point (x_i, y_i), we define g_i to be the distance from (x_i, y_i) to the farthest corner of its cell. The value of g_i is not the same for all sampling points in general. We'll call g_i the "grid spacing" at the sampling point (x_i, y_i).

The global maximum (x_{max}, y_{max}) is in one of these cells, as they span the whole search space. Then, (x_{max}, y_{max}) is within g_i of (x_i, y_i). Because the gradient of $d_{k+1}(x, y)$ always has magnitude 1, $d_{k+1}(x, y)$ can't have changed by more than g_i between (x_{max}, y_{max}) and (x_i, y_i). So, there must be a sampling point whose center is within g_i of the global maximum's center and whose value $d_{k+1}(x_i, y_i)$ is within g_i of $d_{k+1}(x_{max}, y_{max})$. The largest sampling point will have a value closest to the global maximum, by definition, but one of the other top sampling points may have a position closer to the global maximum position. Similar reasoning establishes that the nearest-neighbor interpolation provides a good approximation for $d_{k+1}(x, y)$ across the rest of the search space.

Performance Analysis

The algorithm is an approximation algorithm, and the performance depends on the desired error. Let A be the area of the search space (the area of the Earth), let g be the average grid spacing, and let N be the total number of input points. The number of sampling points is $O(A/g^2)$, and the time required to evaluate $d_{k+1}(x, y)$ at every sampling point is is $O(k \log N\ A/g^2)$. If k is O(N) (if it's a fixed fraction of N), then that becomes $O(k \log N\ A/g^2)$.

The time scales nearly linearly with N, so this algorithm scales well for large data sets. The time scales linearly with A, so if we changed our code to inspect a smaller area the performance would improve accordingly. The error factor may be problematic if high precision is required.

The space required is $O(N + A/g^2)$. This is the bottleneck for precision. Using a method for generating sampling points that only requires constant memory would improve this to O(N).

GUI Design

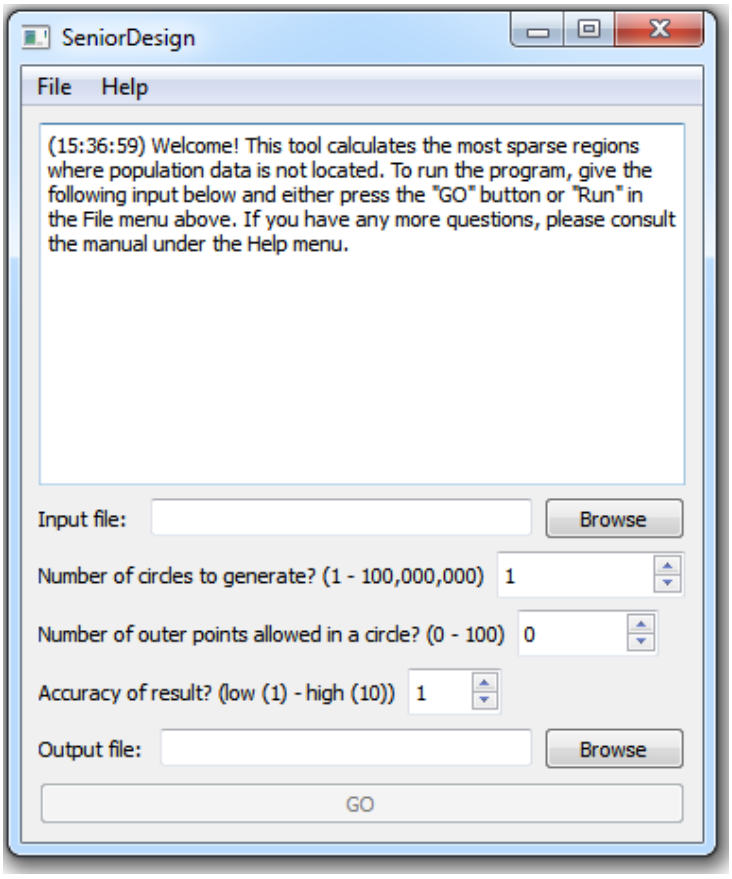

Fig. 5: GUI Screenshot

The design of the GUI application focused on creating a lightweight, multiplatform, and user-friendly interface to make it simple for our client to interact with the algorithm. We used the Qt library because it satisfied these needs and allowed us to quickly produce a way to interact with the algorithm. We chose to separate the algorithm module from the graphical user interface to allow for the algorithm to be runnable in its entirety without a graphical interface.

Implementation details

Languages & Libraries

During the course of this project, we worked with many different programming languages and libraries to complete our task. C++ was the main language utilized. This programming language provided us with the basic tools necessary to run our main project in its entirety. The core library in which the algorithm was built upon was CGAL or the Computational Geometry Algorithms Library. CGAL saved us from the time-consuming process of needing to create their extra backend algorithms which could have been potentially more buggy. The algorithm, implemented with CGAL, was originally written using python for prototyping since one of our members knew this language quite well. The CGAL library runs in both python and C++, so after some kinks were worked out, we transferred the code over to C++ to make it more robust and allow our GUI application as well as any other users or modules to run it. Qt is the C++ library we determined would work great to create the GUI application.

Testing Process

Unit Testing

Some of the basic functions are tested for correctness. For example, functions for moving between a planar map and a sphere are tested to check that they are inverses of each other.

Algorithm Testing

Our algorithm has been tested for correctness on small inputs by comparing results with that of an optimal algorithm (discussed in Appendix II). The algorithm outputs the sampling point positions, the value of $d_{k+1}(x, y)$ at each sampling point, and the grid spacing around each of these. Several tests have verified that the highest sampling point is within the tolerance indicated by its grid spacing of the global maximum.

Performance tests have been performed. The largest number of geodesic grid subdivision iterations we could achieve is 10. This produces 10,485,762 sampling points, and causes the program to peak at about 8 GB of memory usage. Running this on a randomly generated data set with N=1,000,000 input points and k=1,000 took 45 minutes on one of our computers. The output is sorted by radius, descending. Here's the first few lines of the output for that test case:

Latitude	Longitude	Radius (m)	Grid spacing (m)
3.3694030	-165.1577028	528304.0110501	4370.4272605
3.3701742	-165.2230401	527766.1398303	4371.1253014
-7.5518313	-1.7331102	527486.3614580	4390.6032108
3.3086040	-165.1926503	527411.5997402	4370.7009520
3.3093587	-165.2579801	527360.3486624	4371.3948051
-7.7376293	-1.8329856	527272.5754768	4390.4780559
1.6433310	150.0065616	527179.1483774	3617.1577611
-3.3504708	121.3624331	527115.6345737	4394.8683768
-7.7377402	-1.7674941	527096.1302746	4390.4134684
-3.7367498	-114.8439781	527085.7350453	4379.2457942

A basic analysis with R gives:

Latitude	Longitude	Radius (m)	Grid spacing (m)
Min. :-90.00	Min. :-180.0000	Min. : 19838	Min. :1908
1st Qu.:-30.07	1st Qu.: -90.0000	1st Qu.:410168	1st Qu.:4355
Median : 0.00	Median : 0.0000	Median :468971	Median :4388
Mean : 0.00	Mean : 0.0113	Mean :441469	Mean :4336
3rd Qu.: 30.07	3rd Qu.: 90.0000	3rd Qu.:495484	3rd Qu.:4397
Max. : 90.00	Max. : 180.0000	Max. :528304	Max. :4402

Here we see that the positions of the grid points are well distributed all over the Earth, that the radius values are not outrageous (much lower than the circumference of the Earth, etc.), and that the maximum grid spacing for 10 subdivisions is about 4.5 km.

Appendix I: Operation Manual

How to run

1. Unzip the correct file for the particular operating system (i.e. Windows, MacOS, Linux)
2. Run the main program “SparseDetection.exe”
3. Once application loads, you will see an output log with a greeting message.
4. Select the location of the input file which holds the latitude, longitude observation data.
5. Next, select each option to determine how the algorithm with run.
    a. “Number of circles to generate?” signifies how many of the most sparse areas you want to have generated. (Note: Can only generate from 1 to 100 million circles)
    b. “Number of outer points allowed in a circle?” signifies how many points you would like to allow within a sparse region. (Note: Due to efficiency concerns, you can only have up to 100 point within these circles)
    c. “Accuracy of result?” signifies how accurate you want your result to be to the exact solution. The more accurate you want your results, the longer and more memory it will take our approximation algorithm in order to reach that resolution of grid points. (Note: Due to efficiency concerns, we have a designated level from 1 to 10 that allows the user to determine the level of accuracy with 1 being the least accurate and 10 being the most accurate.)
6. Browse for an output file to place the results.
7. At this point, the “GO” button should be enabled. Press the “GO” button to run the algorithm.
8. While the algorithm is running, it can be cancelled at any time by pressing the “Cancel” button on the processing dialog.
9. Results of whether the algorithm ran successfully or not will be displayed in the log above the input file box.
10. If the algorithm ran successfully, there will be an output file in the specified location with all the results. Each line is sorted by radius and will look like (“latitude” <space> “longitude” <space> “radius” <space> “grid spacing”).

11. If any issue were to arise, please explore the “Help” tab to answer any questions and provide contact information.

Appendix II: Alternative Designs

(k+1)th degree Voronoi Diagram -- Optimal Algorithm

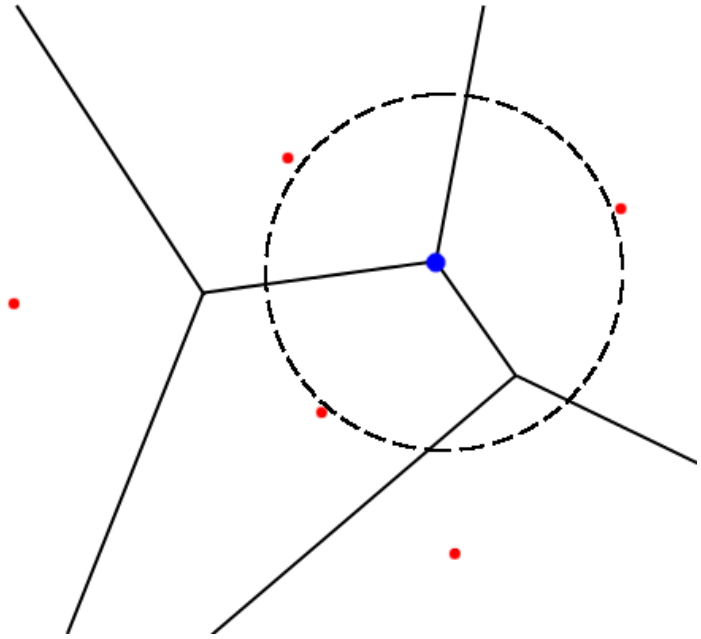

Fig. 6: Local maximum of $d_{k+1}(x, y)$, with k = 0

Our first consideration for implementation involved finding the global maximum by searching through every local maximum. It happens that every local maximum occurs where multiple input points are tied as (k+1)th-closest. These points correspond to vertices on the (k+1)th degree Voronoi diagram constructed from the input points.

This would achieve optimal results, but we could not find any method for computing (k+1)th degree Voronoi diagrams faster than $O(n^2)$, which is be too slow for our purposes. We use a simple version of the optimal algorithm that runs in $O(n^4)$ for validating the approximate algorithm on small datasets. The simple algorithm iterates through every combination of 3 input points, constructs a circle connecting them, and checks that the number of input points inside equals k. This works because every vertex on the (k+1)th degree Voronoi diagram is on the center of one of these circles (and vice-versa)

Iterative Algorithm

One of our first considerations for an approximation algorithm was randomly selecting sample points, locating the k+1 closest point from the sample point, and then slowly moving the sample point from the k+1 point in order to maximize the k-circle. This idea was put on hold because a more straight-forward method for choosing sample points (evenly distributing them) was

decided upon. Even though we did not implement the iterative algorithm, the idea behind it looks promising, and implementing iteration on top of the grid sampling algorithm may result in larger circles.

Appendix III: Other Considerations

Reflection

What We Learned: We learned a multitude of new skills through this entire senior design project. One of which includes how to divide up work based on each team member's individualized skill set. When our group was first created, we did not have a good idea of where anybody else was coming from. This produced a rocky start; however, after collaborating in many team meetings, we were able to persevere past the shyness of the group to better understand how everyone works together. One other big lesson we took away from this project was the importance of how individual knowledge does not equal group knowledge. Scope and communication is the key to completing a project or part of a project. Towards the beginning of this semester, we had our own ideas on how we were going to implement everything. During the building phase, however, we subdivided our group to perform smaller tasks on our own time, and it was evident several times throughout the process that everyone was not on the same page. This communication issue caused some people to double up on work or produce a result that wasn't what was in our specifications. Towards the end of our time as a group, we were able to rally together, and more effectively communicate our ideas to produce our successful product.

Challenges We Faced: Many challenges came and went over the course of this project. One challenge everyone had was learning new libraries and concepts. We had different members in charge of helping design the GUI or building the algorithm or testing the system, but everybody had to work with a new setup which was difficult to initially build on some of our personal machines. The Qt library, used for our GUI, seemed fairly straight forward, but we ran into several bumps and hiccups over where certain files need to be placed in order to run the program, or how to access certain functions and behaviors. Even though we face those hurdles, we were able to pull through with teamwork and a little bit of patience.

References

Lee, D. (). On k-Nearest Neighbor Voronoi Diagrams in the Plane. *IEEE Transactions on Computers*, , 478-487.

Edelsbrunner, H. (). An Improved Algorithm for Constructing kth-Order Voronoi Diagrams. *IEEE Transactions on Computers*, , 1349-1354.

Randall, D. A. (). Numerical Integration of the Shallow-Water Equations on a Twisted Icosahedral Grid. Part I: Basic Design and Results of Tests. *Monthly Weather Review*, , 1862-1880.

Mazumdar, S. (). Fast Algorithms for Computing the Smallest k-Enclosing Circle. *Algorithmica*, , 147-157.

Toussaint, G. T. (). Computing largest empty circles with location constraints. *International Journal of Computer & Information Sciences*, , 347-358.