Florida State University Libraries

Electronic Theses, Treatises and Dissertations

The Graduate School

2012

Instant Approximate 1-Center on Road Networks via Embeddings

Bradley S. Neff

THE FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

INSTANT APPROXIMATE 1-CENTER ON ROAD NETWORKS VIA

EMBEDDINGS

By

BRADLEY S. NEFF

A Thesis submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Master of Science

Degree Awarded:
Spring Semester, 2012

Bradley S. Neff defended defended this Thesis on March 29, 2012.

The members of the supervisory committee were

Piyush Kumar
Professor Directing Thesis

Ashok Srinivasan
Committee Member

Xin Yuan
Committee Member

The Graduate School has verified and approved the above-named committee members, and certifies that the Thesis has been approved in accordance with university requirements.

ACKNOWLEDGEMENTS

I would like to thank my major professor, Dr. Piyush Kumar, and Samidh Chatterjee, for their assistance and guidance in the course of this research. In addition, I would like to thank the members of my committee. Finally I would like to acknowledge the support of the faculty of the Florida State University Department of Computer Science.

— Bradley Neff

TABLE OF CONTENTS

List of Figures	v
Abstract	vii
1. INTRODUCTION	1
2. PROBLEM DEFINITIONS	5
3. EXACT 1-CENTER SOLUTION	6
4. THE ALGORITHM	8
5. PREPROCESSING PHASE	10
5.1 MultiDimensional Scaling	10
5.2 Sammon Optimization Criterion	11
5.3 Handling Large Data Sets	18
6. QUERY PHASE	22
7. INTERFACE	24
7.1 Installation	25
7.2 How To Use	25
7.3 Client	25
7.4 Flask	26
7.5 Query Solvers	26
8. EXPERIMENTS	28
8.1 Experimental Results	28
9. CONCLUSION AND FUTURE WORK	36
APPENDICES	37
A. MEETUP USER MANUAL	37
REFERENCES	45
BIOGRAPHICAL SKETCH	49

LIST OF FIGURES

1.1	Example of a road network with exact (“E”) and approximate (“A”) 1-center computed. The shortest path from the locations to the exact center are also shown. The approximate answer is the one computed using the algorithm presented in this thesis.	2
1.2	An example k-center solution, shown on our interface.	2
1.3	An example k-median solution, shown on our interface.	3
1.4	An example k-mean solution, shown on our interface.	3
3.1	Shortest path plots of query points q_1, q_2, q_3, q_4 along the edge e	6
5.1	Mean distortion obtained for the Sammon and Strain criteria on Tallahassee road network data. Only the distortion for Sammon is reported, because the same for Strain was larger than 10^3 . The data set has 1597 nodes and 2425 edges.	13
5.2	Comparison of Sammon and Strain criteria in terms of time taken on Tallahassee road network data. The time for Strain is not shown after the 7th dimension because its execution time did not change significantly beyond the 7th dimension.	13
5.3	Comparison of distortions from SDPT-3 and Sammon.	15
5.4	Comparison of mean distortions from SDPT-3 and Sammon.	15
5.5	Two examples of 4 point metric spaces that cannot be embedded isometrically into any Euclidean space. Each edge in both the graphs is of unit length. . .	16
5.6	Distortion vs K-Nearest Neighbors	16
5.7	Comparison of distortions from PC and Sammon.	17
5.8	Comparison of mean distortions from PC and Sammon.	18
6.1	A GEQ example. The square points are the points in P and the round ones are the query points. Square point c is the answer to the query.	23

7.1	An interface screenshot. The approximate 1-center to the query points A, B, C, and D is shown.	24
7.2	An overview of data flow within the application. Passed data consists of sets of latitude/longitude pairs.	27
8.1	<i>AbsR</i> values for 10,000 pairwise distances sampled randomly from Tallahassee.	30
8.2	<i>AbsR</i> values for 10,000 pairwise distances sampled randomly from Orlando.	31
8.3	<i>AbsR</i> values for 10,000 pairwise distances sampled randomly from Tampa. .	31
8.4	<i>AbsR</i> values for 10,000 pairwise distances sampled randomly from Florida. .	32
8.5	<i>AbsR</i> values for 10,000 pairwise distances sampled randomly from Miami. .	32
8.6	<i>AbsR</i> values for 1000 pairwise distances sampled randomly from the USA. .	33
8.7	The error in approximation of the 1-center answer computed by the proposed algorithm for Tallahassee.	33
8.8	The error in approximation of the 1-center answer computed by the proposed algorithm for Orlando.	34
8.9	The error in approximation of the 1-center answer computed by the proposed algorithm for Tampa.	34
8.10	The error in approximation of the 1-center answer computed by the proposed algorithm for Florida.	35
8.11	The error in approximation of the 1-center answer computed by the proposed algorithm for Miami.	35

ABSTRACT

This thesis studies the 1-center problem on road networks, an important problem in GIS. Using Euclidean embeddings, and reduction to fast nearest neighbor search, an approximation algorithm is devised for this problem. On real world data sets, extensive experiments are conducted that indicate fast computation of constant factor approximate solutions for query sets much larger than previously computable using exact techniques. A web interface was provided for generic queries on a map interface, with implementations of 1-center, 1-median, 1-mean, k-center, k-median, and k-mean currently in use.

CHAPTER 1

INTRODUCTION

This thesis looks at the 1-center problem on road networks: Compute a ball of minimum radius enclosing a given set of locations (latitude, longitude) on the map. The measurements on the network are done using the shortest path metric. The 1-center problem on road networks has many important applications including location-based services and facility location. Important questions like: where should we construct a fire-station/hospital to cater to a set of people? can be answered by solving the 1-center problem on road networks. Another motivation to study the problem is: given locations of people who want to meet, but want to keep the maximum distance traveled by any one person to a minimum. Figure 1 shows an exact solution to the 1-center problem.

The 1-center problem itself is a subset of the k-aggregate nearest neighbor (kANN) search problem [1]. Given a set P of interesting objects, a set Q of query points, and an aggregate function f (e.g., sum , max) a k-aggregate nearest neighbor (kANN) query retrieves the objects p in P , such that $|P| = k$, $f\{min(d(p_i, q_j), \forall p_i \in P), \forall q_j \in Q\}$ is minimized. For the 1-center problem, the set P is continuous (i.e. it is the whole road network instead of some interesting set of discrete points, as in [1]), $k = 1$, and $f = max$. 1-sum, on which preliminary work has been finished, would have $f = sum$, and would be analogous to minimizing the total fuel consumption for a group of vehicles in the second problem example given above. For the first problem example, if there are 2 hospitals to be built, then k-center with $k = 2$ would minimize the distance any one citizen would have to travel to one of the hospitals. For 1-center, there is an important distinction to be made: problems in which the center can be located anywhere on the road network, and problems in which centers can only be located on the nodes of the network. The former category of problems are known as *absolute center problems*, whereas the latter are known as *vertex center problems*. In

Figure 1.1: Example of a road network with exact (“E”) and approximate (“A”) 1-center computed. The shortest path from the locations to the exact center are also shown. The approximate answer is the one computed using the algorithm presented in this thesis.

Figure 1.2: An example k-center solution, shown on our interface.

this thesis, we implemented an exact absolute 1-center as well as an exact vertex algorithm and compare each of them with an approximation algorithm that produces a vertex center as a solution. Note that using Steiner points on the edges, vertex centers can be used to approximate the absolute center solution easily.

The problem of 1-center computation has a long history and was solved exactly by Kariv and Hakimi [2] in 1979 using $O(n^3)$ time for a road network with n nodes (intersections). Henceforth, this algorithm will be referred to as the KH algorithm. Although algorithmic improvements in shortest path computations in the past two decades [3, 4, 5, 6, 7] could be

Figure 1.3: An example k-median solution, shown on our interface.

Figure 1.4: An example k-mean solution, shown on our interface.

used to improve this running time slightly, it would still be a far cry from an instantaneous solution. One large motivation for trying to solve the problem instantly (even though approximately), is that this could be put on the web for people to compute their results interactively.

Since this is an old and well studied problem, the reader is referred to books that discuss this problem [8, 9]. If m is the number of query locations, it is easy to see that $\Omega(m)$ is a lower bound on this problem since without looking at the queries, its impossible to come up with a correct solution (unless other, perhaps unrealistic assumptions are made). As far as we know, no solution for this problem that works in $O(m + \log n)$ time exists, even for approximate solutions. This thesis demonstrates an approximation algorithm that follows these bounds and gives reasonable approximations. The most interesting aspect of

this solution is that it can cast the problem on the network to a problem in Euclidean space, a first in this line of work. This opens the door for many other similar and related problems to use these techniques and speed up approximate result computations [2, 10, 11]. Below are summarized the contributions of this thesis:

1. A fast and approximate solution is proposed to the 1-center problem on the road network. To our knowledge, there does not exist any practical method for solving this problem instantly in the literature.
2. Embedding techniques are used to map the road network into high dimensional Euclidean space and empirically show that the Euclidean metric in the embedded space is a good approximation of the road network.
3. Extensive experiments are conducted on different data sets which show that embedding gives us fast and good approximate results.
4. A graphical interface is provided for efficient access to the proposed algorithm for generic 1-center queries.

The remainder of the thesis is organized as follows. In Chapter 2 we define the problem formally and introduce the notation we use. Chapter 3 introduces the exact algorithm for the current problem. Chapter 4 gives a brief overview of the algorithm. Chapter 5 presents the preprocessing phase of our algorithm in detail. Chapter 6 describes the tools we use in the query phase. Chapter 7 gives an overview of the web interface. In Chapter 8 we discuss our experimental setup and show our experimental results. Chapter 9 concludes the thesis with ideas for future work.

CHAPTER 2

PROBLEM DEFINITIONS

The following problem definitions are used as the basis for the research in this thesis.

- **Network:** A network is an *undirected* graph $G = (V, E, w)$ where V is the set of vertices (or nodes) and E is the set of edges, and $w : E \rightarrow \mathbb{R}^+$ associates each edge to a positive real number. This number is the weight or cost of the edge. If $V = \{v_1, v_2, \dots, v_n\}$ and there is an edge $e \in E$ such that v_1 and v_2 are its two endpoints, then e is represented as (v_1, v_2) . Let Q be a set of query points $\{q_1, q_2, \dots, q_n\}$ (e.g., users) on the network. In this thesis, using the longitude and latitude of each vertex, the Euclidean length of each edge is computed. This distance is used as the weight of each edge.
- **1-center problem:** The 1-center problem asks for a point p in the graph, such that the maximum distance of p to all points in Q is minimized, when traveling along the edges. The shortest path distance between two vertices v_i and v_j is denoted by d_{ij} , $\forall i, j = 1, 2, \dots, n$.
- **Euclidean Space:** In this paper, it is assumed that the road network is undirected. This enables embedding the network in a Euclidean space. Euclidean space is denoted by $\mathbb{E}^d$, where d is the dimension of the space and the embedded set of vertices by V' . Let $x_i \in V'$ denote the vertex corresponding to v_i . The Euclidean distance between x_i and x_j is denoted by $d'_{ij} = ||x_i - x_j||$.

CHAPTER 3

EXACT 1-CENTER SOLUTION

The exact solution to the 1-center problem was proposed as the KH algorithm [2]. In the road network graph $G = (V, E)$, consider a set of query points $Q = \{q_1, q_2, \dots, q_m\}$ lying on the road network and an edge $e = (v_1, v_2) \in E$, where $v_1, v_2 \in V$. For each of the two endpoints v_1 and v_2 in e , the algorithm computes the shortest path distance from v_1 and v_2 to $q_i \in Q, \forall i = 1, 2, \dots, n$. It then plots the shortest path distance along the edge e from q_i . The shortest path distance plot along e from q_i may be monotonically increasing, monotonically decreasing or both. See Figure 3. See Figure 3. Each of the plots is thus either a straight line or a union of two straight lines. Then the lowest point in the upper envelope of the intersection of these straight lines and the corresponding location in the edge is obtained. The distance of this location to the corresponding query point is recorded. The process is repeated for every edge and the location that gives the minimum of all the recorded distances is the answer to the 1-center query. This algorithm, although an exact one is not practical to use in large road

Figure 3.1: Shortest path plots of query points q_1, q_2, q_3, q_4 along the edge e .

networks because it involves pairwise shortest path computations for each pairwise distance. In the actual implementation, this procedure is optimized by computing the shortest paths from the query points only, instead of from each of the endpoints of every edge.

CHAPTER 4

THE ALGORITHM

Here is presented a brief overview of the algorithm, which has two phases: the preprocessing phase and the query phase. The preprocessing phase, in turn is divided into two parts, the Graph Generation Phase and the Embedding Phase.

Graph Generation Phase: The datasets were obtained from OpenStreetMap website (<http://cloudmade.com>). The next task is to parse the map files to generate the road network graph, followed by the calculation of the pairwise shortest path distances from each vertex. The distance matrix thus generated is fed into the embedding algorithm for mapping the points into an Euclidean space of a chosen dimension.

However, the actual queries might come from anywhere along an edge of the graph. The problem is that the particular point on the edge from where a query is coming from has no representation in the embedding. This is because it is not a vertex in the graph and thus is not a member of the distance matrix. This issue can be fixed by using the following technique. Using the longitude and latitude of each vertex, its 2-dimensional coordinates are calculated. Each edge is divided into shorter, unit length edges using Steiner points [12]. We calculate the map coordinates (latitude,longitude) of a Steiner point by interpolation from the coordinates of original vertices, and also store the fraction (λ , say) that the Steiner point is along the edge. Finally, we store all vertices and Steiner points in a 2-d nearest neighbor (NN) [13] query structure. We denote this set of 2-d points, consisting of the vertices and the Steiner points, by V_2 . To handle a query, the NN structure is used to find the closest vertex or Steiner point, and then obtain the corresponding location in the embedded space. For a Steiner point, one can use λ to interpolate between the embedded positions of the endpoints of its edge.

In the actual implementation, we remove the nodes with degree 2 to compute the

distance matrix for the embedding. Removing these nodes does not affect the shortest path calculations but reduces the data size. We keep the degree-2 nodes for constructing the 2-d NN structure. The number of these nodes on each edge are sufficient enough to act as Steiner points.

Embedding Phase: In the embedding phase, we embed the graph in a low dimensional Euclidean space using Multidimensional Scaling (MDS) [14, 15, 16, 17, 18, 19, 20]. We denote the set of points in the embedded space by V_{d_e} , where d_e is the dimension of the embedded space. After the embedding is done, we compute another NN structure, this time using the points in V_{d_e} .

Query Phase: The input to the query phase are two NN structures (one from V_2 and the other from V_{d_e}), the query points and a set Q_{d_e} . Initially, $Q_{d_e} = \Phi$. The algorithm executes the following steps in this phase.

- **Step 1:** Find the NN of each query point (pairs of longitude, latitude) from V_2 using the 2-d NN data structure.
- **Step 2:** For each query point, if the NN in V_2 has a representation $q \in V_{d_e}$, add q to Q_{d_e} . If the NN is a Steiner point, interpolate its representation in $\mathbb{E}^{d_e}$ and add the result to Q_{d_e} .
- **Step 3:** Compute the minimum enclosing hypersphere of the points in Q_{d_e} .
- **Step 4:** Find the nearest neighbor of the center of the minimum enclosing hypersphere from the points in V_{d_e} . Report the nearest neighbor's corresponding vertex in the original graph as the answer.

CHAPTER 5

PREPROCESSING PHASE

5.1 MultiDimensional Scaling

Multidimensional Scaling is a family of methods for embedding metric spaces into low dimensional Euclidean spaces. For a detailed review of different embedding techniques, the reader is referred to a survey by Maaten et al. [21] and work by Zhang et al. [22]. The aim of this thesis is not to devise a new embedding algorithm. Rather, it is to show that real world road networks can be embedded into a low dimensional Euclidean space (to a reasonable approximation) and how this embedding can be used to solve the problem of interest by using computational geometric tools. In this section, the theoretical bounds for embedding graphs into Euclidean spaces is discussed, followed by an explanation of the measures used to evaluate an embedding. In the next section, the choice of optimization criterion for the MDS is introduced and empirically established as the reason for this choice.

Bounding Errors for Embedding of Graphs: The problem, embedding road network graphs, seeks a method that finds the minimum dimension a data set can be embedded in while maintaining a fixed distortion bound for every pair of distances. A celebrated theorem of Bourgain [23] states that every graph X with a finite number of vertices embeds into Euclidean space with distortion $O(\log n)$. A fundamental question of graph embedding is whether its metric dimension is related to its *intrinsic* dimension. A good measure of intrinsic dimension is the *doubling constant* of the graph. It can be shown that any n vertex graph X embeds into Euclidean space with dimension $O(\dim(X)/\epsilon)$ and distortion $O(\log^{(1+\epsilon)} n)$, where $\dim(X)$ is the doubling dimension of X [24]. For detailed literature reviews, see [24, 25]. All of these algorithms only provide an upper bound and thus do not rule out chances of good embeddings.

Measure of a Good Embedding: The objective of a good embedding is to preserve the

pairwise shortest path distances in the embedded space as closely as possible. Any measure to judge the quality of an embedding should be able to reflect this objective. In this thesis, two types of measures are used: a scaled measure and an absolute ratio measure.

Scaled Measure: This measure uses a scaling factor to compute the distortion. Given $D \geq 1$, if there exists a number $r > 0$, such that

$$r.d_{ij} \leq d'_{ij} \leq D.r.d_{ij}, \quad i, j = 1, \dots, n. \quad (5.1)$$

then the embedding is defined to be a D – *embedding* [26]. The infimum of the numbers D such that the inequality holds is called the distortion of the embedding. r was picked to be the minimum over all $d_{ij}/d'_{ij}, i, j = 1, \dots, n$. The distortion is thus the maximum over all $d'_{ij}/(d_{ij}r)$. Then computed was $d'_{ij}/(d_{ij}r)$ for each (i, j) pair. The average of the numbers obtained is referred to as “mean distortion”. A good embedding should have both the distortion and its mean to be as low as possible. This was the aim when choosing the optimization criterion for MDS.

Absolute Ratio Measure: Apart from the above measure which involves scaling the distances by r , another function is defined which measures the difference between the absolute values of the pairwise distances obtained from the actual graph and the embedding. This helps with evaluating the embedding from a set of pairwise distances sampled from the graph, especially when the size of the graph is too big to compute the actual distortion. In this thesis, this measure is referred to as “*AbsR*” (abbreviation for absolute ratio) and defined as follows.

$$AbsR = \max(d_{ij}/d'_{ij}, d'_{ij}/d_{ij}), \quad i, j = 1, 2, \dots, n. \quad (5.2)$$

Unless otherwise mentioned, the scaled measure is used for calculating distortions.

5.2 Sammon Optimization Criterion

As discussed in the previous section, an optimization criterion was sought that would minimize both the distortion and its mean. First the optimization criterion used in this work is introduced, and then the choice is empirically justified in the following sections. The “Sammon Projection” [27] is used for the embedding purpose of this thesis. The function is given by

$$S = \frac{1}{\sum_{i < j} d_{ij}} \sum_{i < j} \frac{(d_{ij} - d'_{ij})^2}{d_{ij}} \quad (5.3)$$

The Sammon cost function differs from the classical MDS in the sense that it puts more emphasis on retaining distances that were originally small. The minimization of the stress function can be performed using various methods, such as the eigendecomposition of a pairwise distance matrix, the conjugate gradient method, or a pseudo-Newton method [21]. In this thesis, the conjugate gradient method implemented in MATLAB is used. Sammon mapping adapts the classical cost scaling function by weighting the contribution of each pair of datapoints to the cost function [21] by the inverse of their pairwise distance in the original space. In this way, the cost function assigns roughly equal weight to retaining each of the pairwise distances, and thus retains the local structure of the data better than classical scaling [21]. The classical scaling method of MDS was replaced with Sammon mapping for the embedding purposes of this thesis.

5.2.1 Classical vs. Sammon based MDS

Since the pre-processing is mainly based on MDS, the optimization criterion for MDS was first experimentally settled on. MATLAB 7.11.0.584 was used for this purpose. The distortion of classical MDS (also referred to as Strain criterion) was orders of magnitude larger than Sammon. The mean distortion was also large for Strain compared to Sammon, as Figure 5.1 suggests. One advantage that Strain has compared to Sammon is speed (See Figure 5.2), but since this was pre-processing, and the main aim was to build better embeddings, the choice was clear.¹

From Figures 5.1 and 5.2, it was clear that not only was the distortion low when using 6 dimensions, the computation time for Sammon was also not very high. Hence, for the experiments that follow, the embedding dimension was fixed to 6 whenever the Sammon projection was used. These experiments were repeated on several other road networks, but the results obtained were similar. Hence, they are not reported here.

5.2.2 Optimal Embedding for Road Network Graph

In this section it is shown how to produce an optimal embedding using the measure of distortion, described in this thesis, using a semidefinite program (SDP). The practical

¹Two other optimization criteria for non-classical MDS, namely the “metricstress” and the “metricsstress” are available in MATLAB. During experimentation, these functions produced slightly worse results than Sammon as far as distortion and mean distortion are concerned. Also, speed of convergence was similar to that of Sammon. Hence, neither of these two functions were selected.

Figure 5.1: Mean distortion obtained for the Sammon and Strain criteria on Tallahassee road network data. Only the distortion for Sammon is reported, because the same for Strain was larger than 10^3 . The data set has 1597 nodes and 2425 edges.

Figure 5.2: Comparison of Sammon and Strain criteria in terms of time taken on Tallahassee road network data. The time for Strain is not shown after the 7th dimension because its execution time did not change significantly beyond the 7th dimension.

limitations of this method are then stated. The program follows:

$$\begin{aligned}
\text{P1 : Minimize } & \sum_{ij} \epsilon'_{ij} \\
\text{s.t. } & (1 - \epsilon'_{ij})d_{ij} \leq \|x_i - x_j\| \leq (1 + \epsilon'_{ij})d_{ij}; \\
& \epsilon'_{ij} \geq 0; \quad i, j = 1, \dots, n.
\end{aligned}$$

To feed the program to an **SDP** solver, it is modified as follows. Squaring and expanding the first constraint gives: $(1 - \epsilon'_{ij})^2 d_{ij}^2 \leq (x_i x'_i - 2x_i x_j + x_j x'_j) \leq (1 + \epsilon'_{ij})^2 d_{ij}^2$. Replacing $(1 - \epsilon'_{ij})^2$ by $(1 - \epsilon_{ij})$, the final form of the semidefinite program becomes

$$\begin{aligned}
\text{P2 : Minimize } & \sum_{ij} \epsilon_{ij} \\
\text{s.t. } & (1 - \epsilon_{ij})d_{ij}^2 \leq (y_{ii} - 2y_{ij} + y_{jj}) \leq (1 + \epsilon_{ij})d_{ij}^2; \\
& \epsilon_{ij} \geq 0; \\
& y_{ij} \geq 0; \quad i, j = 1, \dots, n.
\end{aligned}$$

In the above program ϵ_{ij} and y_{ij} are (i, j) th elements of the error matrix $\boldsymbol{\epsilon}$ and the Gram matrix $\mathbf{Y}$ of the embedded points, respectively. SDPT-3 [28, 29] was used to solve P2. However, as observed in Cayton et. al [30], general purpose **SDP**-solvers are too memory intensive for even modestly sized programs. For this reason, in these experiments, the data set had to be restricted to $n \leq 100$ points.

The performance of the Sammon function is compared with that of **SDP** in Figure 5.3 and 5.4 on five different subgraphs chosen randomly from the map of the USA. It is to be noted that SDPT-3 embeds in n dimensions while Sammon embeds in 6 dimensions. Reducing the dimension of the embedding by SDPT-3 will reduce the gap between the methods in the graphs. The abbreviations on the x-axes of the graphs show the names of the states from which the subgraphs came from.

Non Embeddable Metric Spaces: There exists simple examples of metric spaces which are not isometrically embeddable into any Euclidean space [26]. Two such examples, given in figure 5.2.2, are often encountered as subgraphs in road networks. **SDP** as well as the Sammon projection was used to embed these graphs. Both the methods produced the exact graphs in the 2-dimensional Euclidean space, thus amounting to the same distortion.

5.2.3 SDP with k-Nearest Neighbors

Techniques like LLE [31] or IsoMap [32] try to embed high dimensional data to low dimensions using information from the k -nearest neighbors (k -NN) of each data point. Empirically, it was attempted to find a k such that using the pairwise shortest path distances

Figure 5.3: Comparison of distortions from SDPT-3 and Sammon.

Figure 5.4: Comparison of mean distortions from SDPT-3 and Sammon.

of only the k -nearest neighbors of each point, it would be possible to get an embedding with minimum distortion. For this purpose, SDP was reformulated to the following form.

$$\begin{aligned}
 \text{P3: Minimize } & \sum_{ij} \epsilon_{ij} \\
 \text{s.t. } & (1 - \epsilon_{ij})d_{ij}^2 \leq (y_{ii} - 2y_{ij} + y_{jj}) \leq (1 + \epsilon_{ij})d_{ij}^2; \\
 & \epsilon_{ij} \geq 0; \\
 & y_{ij} \geq 0; \quad i = 1, \dots, n, \quad j = v_1, \dots, v_k (k < n).
 \end{aligned}$$

Figure 5.5: Two examples of 4 point metric spaces that cannot be embedded isometrically into any Euclidean space. Each edge in both the graphs is of unit length.

where v_l , $l = 1, \dots, k$ denote the l -th nearest neighbor of the vertex v_i . For each point, SDPT-3 was restricted to check for nk constraints, instead of n^2 . Using this SDP, the subgraphs used in the previous section were embedded again. For each k , the distortion obtained was collected. One of the observations is presented in Figure 5.6. In the graph, the smallest k

Figure 5.6: Distortion vs K-Nearest Neighbors

for which the distortion is lowest, is very close to n . A similar trend was observed for mean distortion. For each of the subgraphs that were embedded, the observations were similar. Due to this reason, the route of embedding the graphs using k-NN based methods like LLE or IsoMap was not pursued.

5.2.4 Sammon vs Place Center MDS

The place center (PC) algorithm [33] is a very recent embedding technique proposed by Agarwal et al. It is a unified algorithmic framework for solving many variants of MDS, although the Sammon function does not fall into the family of error functions that PC optimizes. The approach is based on an iterative local improvement method. The algorithm picks a point and moves it so that the cost function is locally optimal and repeats this process until convergence. PC was run on five different subgraphs from the map of the USA. It is to be noted that the size of these subgraphs is much larger (≈ 1500) than those used for the experiments in the previous section. In fact, the subgraphs for doing these experiments were chosen first, and subset of each subgraph was used to generate graphs of smaller sizes for the experiments with SDP. Thus, the names of the states in these experiments remain the same as before. The dimension of embedding used here is 6. The experiments showed that the distortions of the embeddings produced by PC did not change by much after the 5th dimension and continued to be worse than Sammon. In Figures 5.7 and 5.8 the results obtained by optimizing the fMDS cost function [33] are presented. The observations were similar when rMDS [33] and r^2 MDS [33] were optimized. From the figures, it is clear that Sammon is a better choice.

Figure 5.7: Comparison of distortions from PC and Sammon.

Figure 5.8: Comparison of mean distortions from PC and Sammon.

5.3 Handling Large Data Sets

So far the reason for choosing the Sammon function as the optimization criterion has been established. However, all the above results were pertaining to small data sets, of size ≈ 1500 . For large data sets, this procedure becomes extremely memory intensive, because of the number of variables the function has to optimize. Several approximations to classical MDS have been proposed to address its poor scalability. One class of algorithms is based on a spring-mass model of Chalmers et al. [14, 17]. Williams et al. have suggested an improvement to Chalmers' approach, and added user control to speed up performance [20]. A disadvantage of spring-based models, in general, is that they are subject to local minima, and that they require an *a priori* assumption of the data set's underlying dimensionality [34]. FastMap [15], MetricMap [19], and Landmark MDS(LMDS) [16] approximate classical MDS by solving MDS for a subset of the data and fit the remainder to the solution. Platt, in [35], shows how all three algorithms belong to a class of methods called Nyström algorithms, which approximates the solution to the eigenproblem of a large matrix and concludes that LMDS is the fastest and most accurate of the three. One disadvantage encountered when using these algorithms is that all of these techniques require the entire distance matrix to be supplied as one of their inputs. Generating the distance matrix is expensive both in terms of computation and storage, especially for the described problem, where the number of vertices in the road

network graph goes up to roughly 27 million for the entire map of the USA. Even if one generates the whole matrix and applies the exact algorithm to find the 1-center, it will involve $O(mk)$ memory lookups during the query time, where m is the number of edges in the graph and k is the number of query points. Clearly, this can be expensive if m and/or k is very high.

In this thesis, the sampling used is based MDS(referred to as **FastMDS**) proposed in Yang et. al [34]. In the next section an overview of **FastMDS** is given, followed by the proposed algorithm for handling large data sets.

5.3.1 Sampling Based FastMDS

The **FastMDS** approach is based on the observation that a submatrix along the diagonal of a dissimilarity matrix is itself a dissimilarity matrix. Instead of running **MDS** on the full $n \times n$ matrix M , it is partitioned along the diagonal into p submatrices $M_1, M_2, \dots, M_p$, each of size $n_p \times n_p$. A submatrix will be referred to as M_i , where $1 \leq i \leq p$. The **MDS** solution is then computed for each submatrix M_i . These individual **MDS** solutions are then stitched together by sampling s points from each submatrix M_i and putting them into an alignment matrix N_{align} of size $sp \times sp$. In principle, s should be at least 1+ the estimated dimensionality of the dataset. In practice, s is oversampled by a factor of 2 or more, to ensure that the data's inherent dimensionality is captured. **MDS** is then run on N_{align} to get d_{low} -dimensional coordinates for the sampled points. There are now two **MDS** solutions for each of the sampled points; one from performing **MDS** on M_i and one from performing **MDS** on N_{align} . The next step is to compute an affine mapping between these two sets of solutions to align them up in a common coordinate system. This is a linear least squares problem:

$$E_{N_{align}} = \mathbf{U} * E_{M_i} + \mathbf{V} \quad (5.4)$$

where E_{M_i} is the **MDS** solution for the sample points from M_i and $E_{N_{align}}$ is the solution from N_{align} that corresponds to sampled points from M_i . Solving for $\mathbf{U}$ and $\mathbf{V}$ gives a mapping between M_i and N_{align} , which is then applied to the rest of M_i to get d_{low} -dimensional coordinates for all n/p points. The process of partitioning the submatrices is applied recursively, until the size of each submatrix is sufficiently low to run **MDS** on. The stopping condition is found as follows. Let $l \times l$ be the largest matrix that allows **MDS** to be executed efficiently. There are two issues that impact the performance of **FastMDS** on an

$n \times n$ matrix. (i) the size of M_i after subdivision and (ii) p , the number of submatrices that are to be stitched together at each conquer step. Ideally, the size of each submatrix after division should be as large as possible without exceeding $l \times l$. By the same token, the size of the alignment matrix N_{align} should also be bounded by $l \times l$. The number of submatrices to be stitched together, p , should be the largest number such that $sp \leq l$.

Assuming M can be embedded in d_{low} dimensions, regular MDS guarantees an optimal embedding in $\mathbb{R}^{d_{low}}$ space. So there exists a d_{low} -dimensional coordinate for each point represented in M . If enough points are sampled from each submatrix M_i , the subset of points in N_{align} should be able to capture the structure in the original point set.

The running time for the entire algorithm is $O(n \log n)$ [34] where n is the number of points in the input.

It is clear from the above overview that FastMDS does not use information about the pairwise shortest path distance between two points if they are in different partitions, unlike algorithms like LMDS, which requires the pairwise shortest path distance between non-landmark points [16] to produce the embedding. That being said, the proposed algorithm for the preprocessing phase for generating the embedding is presented formally. The method for the query phase does not get affected by large data sets.

5.3.2 Complete Preprocessing Phase

The input to the preprocessing phase is the road network map and two scalars: μ and ν . The following steps create a partition tree of the graph with at most μ nodes at each level.

Step 1: Generate the graphs (with and without the degree 2 nodes) from the road network. Generate the 2-d NN structure using the one having degree 2 nodes. Perform the following steps with the other graph.

Step 2: Binary Partition: Select a random node, and find the node furthest from it. This is the first node. Then select the furthest node from that node. This is the second node. For each other node, if it is closer to the first node, put it in the first partition. Otherwise, it is closer to the second node, and put it in the second partition.

Step 3: Repeat **Step 2** on the largest resulting subpartition until the graph is broken into

μ subpartitions, or each partition is less than or equal to size ν . A random sample of size $\frac{\nu}{\mu}$ is taken from each subpartition for later use.

Step 4: Repeat **Step 3** on each of the μ resulting subpartitions until all the subpartitions are of size less than or equal to ν .

Step 5: Embed each subpartition and each sample partition using MDS.

Step 6: Use FastMDS to embed the original graph.

For large graphs, it was found that $\mu = 20$ and $\nu = 1500$ worked well. For graphs smaller than 3000 nodes, MDS was used to compute the embedding.

CHAPTER 6

QUERY PHASE

In this chapter are explained the steps in the query phase. The first step involves **NN** computation. One of the most successful methods for Euclidean **NN**-search is to partition the entire point set using a tree type data structure. Queries are conducted by dropping a search ball on the tree and conducting searches of those nodes which intersect the ball. Tree based nearest neighbor searches have been found to work very well in practice in low dimensions; expected running time can be found to be $O(\log n)$ for $(1 + \epsilon)$ -approximate solutions [36, 37]. A z-order based quadtree nearest neighbor search algorithm is used for implementing the proposed algorithm [13].

The center of the minimum enclosing ball of the query points that is computed in step 3, is essentially the 1-center of the minimum enclosing ball of the points in Euclidean space. Let Q be a set of points in $\mathbb{E}^d$. Let $B(c, r)$ denote a ball centered at c with radius r . $\text{MEB}(Q)$ is used to denote the minimum enclosing ball for Q . $\text{MEB}(Q)$ is the smallest (measured by r) ball $B(c, r)$ such that $Q \subseteq B(c, r)$. In this context, the operator $\subseteq$ refers to the geometric containment, i.e., every point in Q is fully contained by the ball defined by $B(c, r)$. If the center is fixed at p , $\text{MEB}(p, Q)$ denotes the smallest ball centered at p that encloses all points in Q . Efficient algorithms for computing both exact and approximate **MEBs** are available in literature [38, 39, 40].

Steps 3 and 4 in the algorithm used for locating the approximate 1-center is a well studied problem called the *Group Enclosing Query* in literature [41]. Given a set of points P and a query set Q in $\mathbb{E}^d$, a group enclosing query (**GEQ**) fetches the point $p^* \in P$ such that the maximum distance of p^* to all points in Q is minimized [41]. This problem is equivalent to the **ANN** problem for road networks, described in the introduction. [41] propose a worst case $\sqrt{2}$ approximate solution to this problem, if the exact nearest neighbor of a point can be

calculated. This approximation ratio has been experimentally shown to never exceed 1.05 in practice up to six dimensions. It has a $O(\log n + m)$ query cost in any fixed dimension, where n and m are the sizes of the point set P and the query set Q respectively. The algorithm computes the minimum enclosing ball (MEB) of the query points and then finds the nearest neighbor of the center of the ball from the set P . We state an important theorem regarding their algorithm. For proofs, please refer to Li et. al [41].

Figure 6.1: A GEQ example. The square points are the points in P and the round ones are the query points. Square point c is the answer to the query.

Theorem 6.0.1. *Given a data structure that can do σ -approximate nearest neighbor queries, and another algorithm that can compute ρ -approximate minimum enclosing balls, the GEQ algorithm computes $\sqrt{\sigma^2 + \rho^2}$ -approximate solution.*

In this paper, the exact MEB is computed using CGAL [42] and $(1 + \epsilon)$ -approximate NN of the center of the MEB using STANN [13].

CHAPTER 7

INTERFACE

Figure 7.1: An interface screenshot. The approximate 1-center to the query points A, B, C, and D is shown.

In order to provide easy access to generic 1-center queries, a web interface was developed. A javascript map showing the area available to query is provided to the user (figure 7), and the resulting queries are handled through Flask (a python microframework) and C++ libraries. For selecting query points, a nearest neighbor lookup is performed between the clicked point and the nodes of the network in coordinate space. Note that Steiner points are included at this stage to better match query points on an edge of the network. This

answer is displayed to the user on the map. Also reported (but hidden to the user) are the equivalent non-Steiner points, consisting of the two endpoints of the Steiner point and the distances between the Steiner point and the endpoints. For a 1-center query, the endpoint pairs are sent and converted to points in embedded space. The 1-center is then calculated as described in this thesis. Answers are reported as a latitude/longitude coordinate, which is then plotted on the map. For comparison purposes, the exact answer can also be calculated and displayed. However, this ability is only for comparison testing purposes.

7.1 Installation

Install the package using the command

```
$ python setup.py <install-dir-full-path>
```

This will set up the package in the specified directory and make the required executables.

7.2 How To Use

To start the backend servers, in `/scripts` run

```
python start_all.py
```

To start a local interface instance, in `/interface` run

```
python map.py
```

In the next few sections are provided an overview of the components of the user interface. For a summary, see figure [7.2](#).

7.3 Client

The client is designed in javascript, and utilizes the Google Maps API (version 3), and jQuery. The user's query point and answer data are stored here (not on the server), and are sent to the server upon any calculation request. The map is capable of loading either Google map tiles, or open street map tiles. The current choice can be changed through the selection box in the upper right hand corner of the map. On initialization, the map centers and zooms to a preset location. A bounding rectangle is placed around a designated area of the map

which specifies the bounds of accepted query points. This utilizes the Google Maps rectangle object. Query Points can be placed by clicking on the map, or by entering an address. Such actions result in a query being sent to the back end (covered in more detail in section 7.4). The resulting answer is added to the map upon reception. 1-aggregate queries require at least two valid query points. If less have been provided, the button to request a calculation will be grayed out and unavailable. Once enough query points are available, selecting the "Choose Algorithm" button will provide a drop down menu, allowing for the selection of a particular query. Options for 1-aggregate queries are provided directly, while k-aggregate queries are available under the advanced option. The currently available aggregate functions available are mean, center, and median. k-aggregate queries are handled in a similar manner to Query Point queries, being sent to the back end by a customized url request. 1-aggregate queries are sent as k-aggregate queries with $k = 1$. Depending on the query, either the endpoint pairs or the original nearest neighbor point will be used. The answer is reported on the map once an answer is received. This answer is represented by a uniquely colored marker to differentiate it from query points and other k-aggregate answers.

7.4 Flask

Flask is a web development microframework that utilizes custom urls to route to specific python functions [43]. For Query Point requests, the `find_nearest_neighbor` function unpackages the query received from the client and writes the query files for the C++ backend. The function then waits for the answer files to be written, and packages the result for the client. 1-aggregate functions are handled similarly, through the `find_k_aggregate` function. Besides the query point and one-aggregate query functions, url functions are provided to request various server resources, including javascript files and images (notably, point markers are requested this way). Flask interacts with our web server using `mod_wsgi`, an apache module that supports the WSGI interface [44].

7.5 Query Solvers

Two executables solve the query files provided by Flask. `nearest_neighbor_server` answers the query files provided in `interface/servers/nearest_neighbor_server/input`, and reports the answer in `interface/servers/nearest_neighbor_server/output`. A nearest

neighbor calculation is performed on the query point within the query file, with the answer point being a point within the map. Answers consist of two sets of points: a pair of values corresponding to the latitude and longitude of a point within the original map, and two pairs of values giving the latitudes and longitudes of embedding points that are connected to the original map point. The distance along the edge is also provided.

`embedding_server` handles all the k-aggregate queries that Flask generates. It answers the query files provided in `interface/servers/embedding_server/input`, and reports the answer in `interface/servers/embedding_server/output`. Answers consist of a series of pairs of numbers, with each pair corresponding to the latitude and longitude of an answer point.

Figure 7.2: An overview of data flow within the application. Passed data consists of sets of latitude/longitude pairs.

CHAPTER 8

EXPERIMENTS

The proposed algorithm was implemented in C++ and MATLAB. The implementation of the exact algorithm and the graph partitioning part of the algorithm was written in C++ and compiled with g++ 4.1.2 with -O3 optimization. For constructing the graphs from the road networks, the BOOST [45] library was used. The CGAL [42] and STANN [13] libraries were used for implementing the KH algorithm, computing the Minimum Enclosing Ball and the NN-search in the Euclidean space. FastMDS was implemented using MATLAB. The machine used has 8 Quad-Core AMD Opteron(tm) Processor 8378 with hyperthreading enabled. Each core has a L1 cache size of 512 KB, L2 of 2MB and L3 of 6MB with 128 GB total memory. The operating system was CentOS 5.3. All data was generated and stored as 64 bit C++ doubles. Currently, 24 cores of this machine were used for preprocessing and one core for the queries. Two sets of graphs are presented in the next section. In the first set, it is shown if the embedding that was generated from the graphs is meaningful. To this end, the results are presented from the embedding of the road networks from Tallahassee (1597 nodes and 2425 edges), Orlando (2954 nodes and 4602 edges), Tampa (2932 nodes and 4554 edges), Miami (6554 nodes and 10063 edges), Florida (1001982 nodes and 1327731 edges) and the entire United States (26136771 nodes and 32982669 edges). The number of nodes and edges given are obtained after removal of degree 2 nodes. The second set shows the error in approximation of the 1-center radius obtained from the approximate answer when compared against the exact answer.

8.1 Experimental Results

Once the shortest path metric was embedded into $\mathbb{E}^6$, it was desirable to make sure that the embedding was meaningful. These experiments use the *AbsR* measure, because of the

difficulty of acquiring information for all pairwise distances for the large data sets, which are necessary for computing the scaling factor. Although small sets do not have this problem, the same measure is used for both. 10,000 pairwise distances were sampled from the road networks of the maps mentioned in the previous section. From the map of the USA, 1000 pairwise distances were sampled, since the computation of 10,000 shortest paths was too expensive for the available resources. Figures 8.1, 8.2, 8.3, 8.4, 8.5, 8.6, show the *AbsR* measures for the pairwise distances. Note that most of these pairs have very small *AbsR* measures, although Miami, Florida and USA have higher *AbsR* measures. This is due to the fact that they fall under the category of “large data,” hence **FastMDS** had to be used for their embedding, thus increasing the error factor.

Encouraged by these results, the exact radius computed using KH algorithm and the radius computed by the proposed algorithm were compared. This is shown in Figures 8.7, 8.8, 8.9, 8.10, 8.11. There is a slight abuse of notation in here. There are two comparisons done in these graphs. One is with the “absolute answer” computed by the KH algorithm, shown by “Absolute Distortion” in the graph. The second is with the exact “vertex answer,” shown by “Vertex Distortion.” The term “Mean Radii Distortion” is used for the measuring of the error in approximation of the answer from the proposed algorithm with the true radius. Let R_{ab} and R_v denote the true absolute and vertex radii and R_{approx} denote the approximate radius. All the radii are shortest path ones, i.e. the shortest path distance between the 1-center and the furthest point from it. Then the figures in the y-axes of the graphs are the average from 50 runs of R_{approx}/R_{ab} for computing the Absolute Distortion and R_{approx}/R_v for computing the Vertex Distortion. It shows that the proposed algorithm is closer to the vertex answer than the absolute answer. This is because R_{approx} is greater than both R_{ab} and R_v , and $R_v > R_{ab}$. Hence $R_{approx}/R_v < R_{approx}/R_{ab}$. The increase in distortion of both the answers occur when the queries are fed from the embedded points having large distortions. Worthy of note is that for Miami and Florida, the distortions are higher than those of Tallahassee, Tampa or Orlando. The distortion is always above the 1.1 mark in the case of Miami unlike the other three. This is because Miami and Florida have more points that are badly embedded. This is due to the fact that they were embedded piecewise using **FastMDS**. The timings for both algorithms are shown in Table 8.1. As stated before, a cubic time algorithm is no match to the proposed algorithm.

Table 8.1: Time chart in seconds for computing the 1-center for query sizes ranging from 2 to 2^{10}

		2	2^2	2^3	2^4	2^5	2^6	2^7	2^8	2^9	2^{10}
Tallahassee	Vertex	0.002	0.003	0.005	0.01	0.021	0.04	0.08	0.162	0.336	0.710
	Absolute	0.006	0.013	0.031	0.053	0.132	0.306	0.809	2.052	4.874	11.238
	Approximate	10^{-7}	10^{-7}	10^{-6}	10^{-6}	10^{-6}	10^{-5}	10^{-5}	10^{-4}	1.3×10^{-3}	1.3×10^{-3}
Tampa	Vertex	0.002	0.005	0.010	0.021	0.043	0.087	0.175	0.358	0.721	1.454
	Absolute	0.009	0.0268	0.06	0.134	0.239	0.505	1.252	2.545	5.449	12.839
	Approximate	10^{-6}	10^{-6}	10^{-6}	10^{-6}	10^{-6}	10^{-5}	10^{-5}	10^{-4}	1.4×10^{-3}	2×10^{-3}
Orlando	Vertex	0.003	0.005	0.010	0.021	0.043	0.087	0.175	0.357	0.732	1.462
	Absolute	0.009	0.027	0.063	0.133	0.27	0.538	1.125	2.964	7.113	11.959
	Approximate	10^{-6}	10^{-6}	10^{-6}	10^{-6}	10^{-6}	10^{-5}	10^{-5}	10^{-4}	1.4×10^{-3}	1.4×10^{-3}
Miami	Vertex	0.007	0.014	0.028	0.056	0.112	0.226	0.459	0.923	1.864	3.77
	Absolute	0.021	0.060	0.116	0.259	0.467	1.095	2.162	4.513	13.072	41.818
	Approximate	10^{-4}	10^{-4}	10^{-3}	10^{-3}	10^{-3}	10^{-3}	10^{-3}	10^{-3}	1.5×10^{-3}	3×10^{-3}
Florida	Vertex	4.88	9.855	19.95	39.92	79.44	159.44	321.94	656.38	1309.9	2611.26
	Absolute	7.190	17.007	35.276	68.288	138.7	235.54	434.16	818.44	1467.98	2869.947
	Approximate	0.023	0.024	0.022	0.023	0.027	0.024	0.025	0.037	0.048	0.053

Figure 8.1: *AbsR* values for 10,000 pairwise distances sampled randomly from Tallahassee.

Figure 8.2: $AbsR$ values for 10,000 pairwise distances sampled randomly from Orlando.

Figure 8.3: $AbsR$ values for 10,000 pairwise distances sampled randomly from Tampa.

Figure 8.4: $AbsR$ values for 10,000 pairwise distances sampled randomly from Florida.

Figure 8.5: $AbsR$ values for 10,000 pairwise distances sampled randomly from Miami.

Figure 8.6: $AbsR$ values for 1000 pairwise distances sampled randomly from the USA.

Figure 8.7: The error in approximation of the 1-center answer computed by the proposed algorithm for Tallahassee.

Figure 8.8: The error in approximation of the 1-center answer computed by the proposed algorithm for Orlando.

Figure 8.9: The error in approximation of the 1-center answer computed by the proposed algorithm for Tampa.

Figure 8.10: The error in approximation of the 1-center answer computed by the proposed algorithm for Florida.

Figure 8.11: The error in approximation of the 1-center answer computed by the proposed algorithm for Miami.

CHAPTER 9

CONCLUSION AND FUTURE WORK

This work opens up many new and interesting avenues of research. One immediate open problem is how to reduce the error in approximation of the 1-center answers for large data sets. One can find the lowest common ancestor in the partition tree and use the embedding of that partition to locate the answer. As for the embedding, other methods of embedding are being looked in to, such as proposed by Zhang et. al. [22], and ways of embedding the network into L_1 metric. The initial experiments on small graphs gave encouraging results. As for the semidefinite programming approach, it is to be seen whether semidefinite relaxation techniques [46] will be effective in handling large data sets.

APPENDIX A

MEETUP USER MANUAL

Provided here is the documentation for the meetup interface.

Chapter 1

Dependencies

Running the interface currently requires the following outside dependencies:

Python - <http://python.org/>

CGAL - <http://www.cgal.org/>

Boost C++ Libraries - <http://www.boost.org/>

Flask - <http://flask.pocoo.org/>

mod_wsgi - <https://code.google.com/p/modwsgi/>

Chapter 2

map.py

This module is a Flask application that bridges the gap between the backend query servers and the client. Parameters are recieved over http through custom url extensions unique to each function. Two function types are provided: Dynamic Query Functions and Static Functions.

Chapter 3

Dynamic Query Functions

These functions request dynamic information from the server.

`map.find_k_aggregate()`

Perform the requested calculation on the set of query points, and return the answer.

Uses the url extension ‘/_calculate’

Queries are recieved over http in the following form:

lats_1: list of latitudes that represent the longitude of the first endpoint for each edge a query point is on.

lons_1: list of longitudes that represent the longitude of the first endpoint for each edge a query point is on.

lats_2: list of latitudes that represent the latitudes of the second endpoint for each edge a query point is on.

lons_2: list of longitudes that represent the longitude of the second endpoint for each edge a query point is on.

deltas: list of percentages that represent the distance from the first endpoint to the query point in terms of the total distance of the edge.

type: the type of the calulation. Currently supported are center, mean, and median.

k: maximum number of answers to return.

Answers are returned over http as a latitude, longitude pair in the following form:

points: list of points containing up to k entries in the following form: lat: latitude of an answer point. lon: longitude of an answer point.

`map.find_nearest_neighbor()`

Find the nearest neighbor of a coordinate point server side. Queries are read from http

requests, with the variable 'point' containing the coordinates that represent the latitude and longitude of the point.

Uses the url extension '/_find_nearest_neighbor'

Answers are returned over http in the format: success: boolean value lat: absolute latitude of the answer lon: absolute longitude of the answer lat_approx_1: latitude of the first endpoint of the edge

the absolute answer is on

lon_approx_1: longitude of the first endpoint of the edge the absolute answer is on

lat_approx_2: latitude of the second endpoint of the edge the absolute answer is on

lon_approx_2: longitude of the second endpoint of the edge the absolute answer is on

delta: distance from the first endpoint to the absolute point divided by the length of the entire edge. Represents the location of the absolute answer on the edge as a percentage.

map.**in_current_bounds** (*point*)

Return if the point would lie within the bounding box

Chapter 4

Static Member Functions

These functions are provided to access static elements of the page (images, javascript, etc.).

`map.alg_elements ()`

return the elements of the algorithm list

Uses the url extension `'/_alg_elements'`

`map.fetch_map_style ()`

return the css for the index

Uses the url extension `'/_get_center'`

`map.fetch_marker (color_and_index)`

Return a marker with the specified color and letter, with the letter corresponding to the index modded by 26

Uses the url extension `'/_static/markers/<color_and_index>'` *color_and_index* is a variable of the form *colorname_number*

`map.fetch_marker_number (color_and_index)`

Return a marker with the specified color and letter, with the letter corresponding to the index modded by 26

Uses the url extension `'/_static/markers_with_number/<color_and_index>'` *color_and_index* is a variable of the form *colorname_number*

`map.fetch_overlay_close_javascript ()`

return the javascript for the overlay closer

Uses the url extension `'/_static/overlay-close.png'`

`map.fetch_overlay_javascript ()`

return the javascript for the overlay

Uses the url extension `'/_static/overlay.js'`

`map.fetch_overlay_style()`
return the css for the overlay
Uses the url extension `'/_static/overlay.css'`

`map.get_bounds()`
Returns the jsonified bounding box. Uses the url extension `'/_get_bounds'`

`map.get_center()`
Find and return the lat/lon of the center of the bounding box. Uses the url extension `'/_get_center'`

`map.get_current_path()`
Return the current path

`map.instructions()`
return the help page
Uses the url extension `'/_instructions'`

`map.mobile()`
Return the mobile index page
Uses the url extension `'/mobile'`

`map.root_index()`
Return the main index page
Uses the url extension `'/'`

Chapter 5

meetup.wsgi

This is an example mod_wsgi file that will interface our application with the currently running web server:

```
import sys
sys.path.append("<path_to_application>/meeting_point_installer_2/interface")
from map import app as application
```

For apache, this also requires an entry in the httpd.conf file, consisting of:

```
WSGIDaemonProcess meetup threads=5
WSGIScriptAlias / /var/www/meetup/meetup.wsgi

<Directory /var/www/meetup>
    WSGIProcessGroup meetup
    WSGIApplicationGroup meetup
    Order deny,allow
    Allow from all
</Directory>
```

Note: This assumes that meetup.wsgi is placed in /var/www/meetup/.

This will allow the application to be accessed from the root of the web server. If an extension is required, the first parameter of WSGIScriptAlias should be modified to reflect this.

REFERENCES

- [1] Man Lung Yiu, Nikos Mamoulis, and Dimitris Papadias. Aggregate nearest neighbor queries in road networks. *IEEE Trans. on Knowl. and Data Eng.*, 17:820–833, June 2005. [1](#)
- [2] O. Kariv and S. L. Hakimi. An algorithmic approach to network location problems. i: The p-centers. *SIAM Journal on Applied Mathematics*, 37(3):pp. 513–538, 1979. [1](#), [3](#)
- [3] Christian Sommer. *Approximate Shortest Path and Distance Queries in Networks*. PhD thesis, The University of Tokyo, 2010. [1](#)
- [4] J. A. Moreno. A new result on the complexity of p-center problem. Technical report, Universidad Complutense, 1986. [1](#)
- [5] Arie Tamir. New results on the complexity of p-center problems. *SIAM J. Comput.*, 12:751–758, 1983. [1](#)
- [6] Arie Tamir. On the solution value of the continuous p-center location problem on a graph. *Math. Oper. Res.*, 12:340–349, May 1987. [1](#)
- [7] Arie Tamir. Improved complexity bounds for center location problems on networks by using dynamic data structures. *SIAM J. Discret. Math.*, 1:377–396, August 1988. [1](#)
- [8] M. S. Daskin. *Network and Discrete Location: Models, Algorithms, and Applications*. John Wiley, New York, NY, 1995. [1](#)
- [9] P. B. Mirchandani and R. L. Francis, editors. John Wiley & Sons, 1990. [1](#)
- [10] Sattam Alsubaiee, Alexander Behm, and Chen Li. Supporting location-based approximate-keyword queries. In *Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS ’10, pages 61–70, New York, NY, USA, 2010. ACM. [1](#)
- [11] Yang Li, Feifei Li, Ke Yi, Bin Yao, and Min Wang. Flexible aggregate similarity search. In *Proceedings of the 2011 international conference on Management of data*, SIGMOD ’11, pages 1009–1020, New York, NY, USA, 2011. ACM. [1](#)
- [12] Mark de Berg, Otfried Cheong, Mark van Kreveld, and Mark Overmars. *Computational Geometry Algorithms and Applications*. Springer-Verlag, Berlin Heidelberg, 3 edition, 2008. [4](#)

- [13] Michael Connor and Piyush Kumar. Fast construction of k-nearest neighbor graphs for point clouds. *IEEE Transactions on Visualization and Computer Graphics*, 16(4):599–608, 2010. <http://compgeom.com/~stann>. 4, 6, 6, 8
- [14] Matthew Chalmers. A linear iteration time layout algorithm for visualising high-dimensional data. In *Proceedings of the 7th conference on Visualization '96*, VIS '96, pages 127–ff., Los Alamitos, CA, USA, 1996. IEEE Computer Society Press. 4, 5.3
- [15] Christos Faloutsos and King-Ip (David) Lin. Fastmap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. pages 163–174, 1995. 4, 5.3
- [16] Seunghak Lee and Seungjin Choi. Landmark mds ensemble. *Pattern Recogn.*, 42:2045–2053, September 2009. 4, 5.3, 5.3.1
- [17] Alistair Morrison, Greg Ross, and Matthew Chalmers. Fast multidimensional scaling through sampling, springs and interpolation. *Information Visualization*, 2:68–77, March 2003. 4, 5.3
- [18] John C. Platt. Fastmap, metricmap, and landmark mds are all nystrom algorithms. In *In Proceedings of 10th International Workshop on Artificial Intelligence and Statistics*, pages 261–268, 2005. 4
- [19] Jason Tsong-Li Wang, Xiong Wang, King-Ip Lin, Dennis Shasha, Bruce A. Shapiro, and Kaizhong Zhang. Evaluating a class of distance-mapping algorithms for data mining and clustering. In *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '99, pages 307–311, New York, NY, USA, 1999. ACM. 4, 5.3
- [20] Matt Williams and Tamara Munzner. Steerable, progressive multidimensional scaling. In *Proceedings of the IEEE Symposium on Information Visualization*, pages 57–64, Washington, DC, USA, 2004. IEEE Computer Society. 4, 5.3
- [21] L. J. P. van der Maaten, E. O. Postma, and H. J. van den Herik. Dimensionality Reduction: A Comparative Review. 2007. 5.1, 5.2
- [22] Lei Zhang, Ligang Liu, Craig Gotsman, and Steven J. Gortler. An as-rigid-as-possible approach to sensor network localization. *ACM Trans. Sen. Netw.*, 6:35:1–35:21, July 2010. 5.1, 9
- [23] J. Bourgain. On lipschitz embedding of finite metric spaces in Hilbert space. *Israel Journal of Mathematics*, 52(1):46–52, March 1985. 5.1
- [24] Ittai Abraham, Yair Bartal, and Ofer Neiman. Embedding metric spaces in their intrinsic dimension. In *Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '08, pages 363–372, Philadelphia, PA, USA, 2008. Society for Industrial and Applied Mathematics. 5.1

- [25] Piotr Indyk and Jiri Matousek. Low-distortion embeddings of finite metric spaces. In *Handbook of Discrete and Computational Geometry*, pages 177–196. CRC Press, 2004. [5.1](#)
- [26] Jiri Matousek. *Lectures on Discrete Geometry*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2002. [5.1](#), [5.2.2](#)
- [27] Jarkko Venna and Samuel Kaski. Local multidimensional scaling. *Neural Netw.*, 19:889–899, July 2006. [5.2](#)
- [28] K. C. Toh, M.J. Todd, R. H. Ttnc, and R. H. Tutuncu. Sdpt3 - a matlab software package for semidefinite programming. *Optimization Methods and Software*, 11:545–581, 1999. [5.2.2](#)
- [29] R. H. Tutuncu, K. C. Toh, and M. J. Todd. Solving semidefinite-quadratic-linear programs using sdpt3. *Mathematical Programming*, 95:189–217, 2003. [5.2.2](#)
- [30] Lawrence Cayton and Sanjoy Dasgupta. Robust euclidean embedding. In *Proceedings of the 23rd international conference on Machine learning*, ICML '06, pages 169–176, New York, NY, USA, 2006. ACM. [5.2.2](#)
- [31] Sam T. Roweis and Lawrence K. Saul. Nonlinear dimensionality reduction by locally linear embedding. *SCIENCE*, 290:2323–2326, 2000. [5.2.3](#)
- [32] Joshua B. Tenenbaum, Vin Silva, and John C. Langford. A Global Geometric Framework for Nonlinear Dimensionality Reduction. *Science*, 290:2319–2323, 2000. [5.2.3](#)
- [33] Arvind Agarwal, Jeff M. Phillips, and Suresh Venkatasubramanian. Universal multidimensional scaling. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '10, 2010. [5.2.4](#)
- [34] Tynia Yang, Jinze Liu, Leonard Mcmillan, and Wei Wang. A fast approximation to multidimensional scaling, by. In *Proceedings of the ECCV Workshop on Computation Intensive Methods for Computer Vision (CIMCV)*, 2006. [5.3](#), [5.3.1](#)
- [35] John C. Platt. Fastmap, metricmap, and landmark mds are all nyström algorithms. [5.3](#)
- [36] Sunil Arya, David M. Mount, Nathan S. Netanyahu, Ruth Silverman, and Angela Y. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *J. ACM*, 45:891–923, November 1998. [6](#)
- [37] Timothy M. Chan. Approximate nearest neighbor queries revisited. In *Proceedings of the thirteenth annual symposium on Computational geometry*, SCG '97, pages 352–358, New York, NY, USA, 1997. ACM. [6](#)
- [38] Nimrod Megiddo. Linear programming in linear time when the dimension is fixed. *J. ACM*, 31:114–127, January 1984. [6](#)

- [39] P. Kumar, J. S. B. Mitchell, and A. Yildirim. Approximate minimum enclosing balls in high dimensions using core-sets. *The ACM Journal of Experimental Algorithmics*, 8, 2003. 6
- [40] E. Alper Yildirim. Two algorithms for the minimum enclosing ball problem. *SIAM J. on Optimization*, 19:1368–1391, November 2008. 6
- [41] Feifei Li, Bin Yao, and Piyush Kumar. Group enclosing queries. *IEEE Transactions on Knowledge and Data Engineering*, 99(PrePrints), 2010. 6
- [42] CGAL, Computational Geometry Algorithms Library. <http://www.cgal.org>. 6, 8
- [43] Flask. <http://flask.pocoo.org/>. 7.4
- [44] mod_wsgi. <https://code.google.com/p/modwsgi/>. 7.4
- [45] Boost libraries. <http://www.boost.org>. 8
- [46] Nathan Krislock. *Semidefinite Facial Reduction for Low-Rank Euclidean Distance Matrix Completion*. PhD thesis, University of Waterloo, 2010. 9

BIOGRAPHICAL SKETCH

Bradley S. Neff

Bradley S. Neff was born on the 27th of May, 1987, in Boynton Beach, Florida. In the spring of 2009, he completed his Bachelors degree in Computer Science at Wofford College. Under the advisement of Prof. Piyush Kumar, he obtained his Masters degree in spring of 2012, from the Department of Computer Science at FSU.

Brad's research interests include geometric information systems, parallel programming, and high performance computing.

Brad lives in Spartanburg, SC, with his Fiance Alison Monahan.