Graph Construction Through Laplacian Function Optimization

Cameron Musco

Advisor: Dan Spielman

December 09, 2011

Abstract

I study a variety of topics relating to the graph construction technique proposed in 'Fitting a Graph to Vector Data' (Daitch, Kelner, Spielman, 2009). This technique attempts to construct a graph that optimizes how closely each vertex can be approximated as a weighted average of its neighbors. I describe a slightly modified graph construction technique and explore its effectiveness for classification and regression tasks. I also present some interesting experimental results that seem to confirm the conjecture from 'Fitting a Graph to Vector Data' that the average degree of the constructed graphs is somehow correlated with the underlying dimensionality of our data. I give some ideas about how we can improve learning over our graphs and how we can explore the relationship between the optimization function used to construct the graphs and the optimization functions used in the learning algorithms applied to the graphs. Finally, I present some preliminary work looking at the structure of the optimization problem used to construct our graphs. This work eventually helped lead to a counter-example of the conjecture in the original paper that a data set in general position has a unique optimal graph. I will present the full extent of the work on the uniqueness problem in a forthcoming report with Christopher Musco, with whom I collaborated on this part of the project.

1 Background

Traditionally, in graph based machine learning problems, we are presented with a set of data points that fit a graph structure. Each data point corresponds to a node of the graph and weighted edges connect related points. The goal may be to cluster the data into coherent groups, to use a subset of labeled points to classify the remaining unlabeled points, or to estimate the value of a function over unlabeled points using labeled data.

Graph based learning, however, is not only applicable to data that naturally fits a graph structure. Much recent research has been done into how generic vector data can be used to construct a graph that can then be used for learning on the data. Fitting a graph to vector data may improve learning, especially in a semi-supervised setting, in which the structure of unlabeled data points, in conjunction with the labeled data, may be important in determining our final predictions.

K-Nearest Neighbors (kNN) and ϵ-Neighborhood graphs are easy to construct graphs that are often built over vector data sets, sometimes using a variety of different distance functions. However these techniques have limitations: ϵ-Neighborhood graphs do not adapt well to changing point densities in the data set and KNN graphs limit the degree to which a point can be affected by nodes outside of its nearest neighbors. In 'Fitting a Graph to Vector Data', Daitch, Kelner, and Spielman propose a new graph construction method, based off the idea that a data point should be able to be estimated from its neighbors in the graph. (This paper is cited as [4] throughout the rest of this report.)

The authors seek to build a graph that minimizes the total error, weighted by node degree, arising from estimating each point as a weighted average of its neighbors. Specifically, they seek

edge weights, $\boldsymbol{w}$, that minimize:

$$f(\boldsymbol{w}) = \sum_i \|d_i \boldsymbol{x}_i - \sum_j w_{i,j} \boldsymbol{x}_j\|^2$$

$$f(\boldsymbol{w}) = \|LX\|_F^2$$

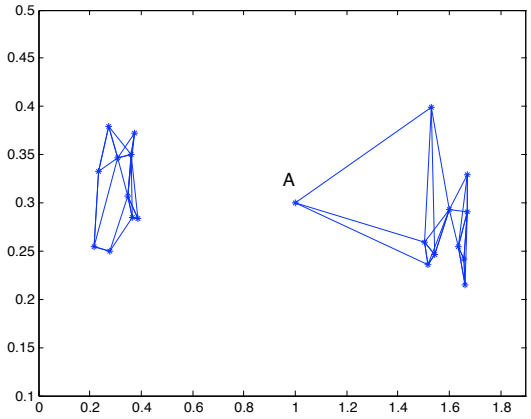
where L is the graph laplacian for the graph with edge weights $\boldsymbol{w}$, and $\|M\|_F = (\sum_{i,j} M_{i,j}^2)^{1/2}$ is the Frobenius norm.

Since setting each weighted degree d_i to 0 would clearly minimize this function, the authors introduce two possible methods for restricting the weighted degrees. In the *hard graph* each weighted degree is required to be at least 1. In the α-*soft graph*, the degrees must satisfy the constraint:

$$\sum_i (max(0, 1 - d_i))^2 \leq \alpha n$$

where α is a chosen parameter of the soft graph.

Figure 1: Failure of KNN graph to capture the fact that A falls between our two clusters. This is the type of failure that the technique proposed in [4] should address.



2 An alternative objective function

For part of my project, I explored the use of a slightly modified objective function to construct graphs over vector data. In the alternative formulation, we minimize $f(\mathbf{w}) = \sum_i \|\mathbf{x_i} - \sum_j w_{i,j}\mathbf{x_j}\|^2$ subject to $w_{i,j} \geq 0 \, \forall \, i, j$. We call this function the *non-degree* function, because, unlike the function used in [4], it does not include the weighting of vertices by their degrees. I explored the performance of graphs built using the non-degree function in combination with various data transformations. I also explored the relationship between the results we achieve using this function and the conjecture by the authors of [4] that our average unweighted graph degrees are somehow related to the underlying dimensionality of the data.

2.1 Motivation and formulation

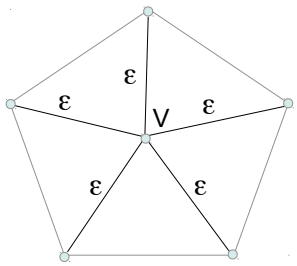
As given above, the original objective function used in [4] is:

$$f(\mathbf{w}) = \sum_i \|d_i \mathbf{x_i} - \sum_j w_{i,j} \mathbf{x_j}\|^2$$

subject to $w_{i,j} \geq 0 \, \forall \, i, j$ (each edge weight must be positive).

This function is trivially minimized if all edge weights are set to zero, giving $f(\mathbf{w}) = 0$. In order to avoid this degeneracy, in the hard graph, each vertex is required to have weighted degree d_i at least 1. i.e. $\sum_j w_{i,j} \geq 1 \, \forall \, i$. In the soft graph, it is required that $\sum_i (max(0, 1 - d_i))^2 \leq \alpha n$, where α is a softness parameter. These degree constraints also prevent graphs in which, to optimize $f(\mathbf{w})$, some weights are set to infinitesimally small $\epsilon > 0$. For example:

Figure 2: V is exactly reconstructed as an average of its neighbors. Weights of the edges to V are set to ϵ as small as possible in order to improve approximation of the external points.



While the above degree restrictions prevent degenerate solutions, they also create some unnatural behavior. The weighting of each component of the objective function by d_i builds a tendency towards uniformly lower degree into the function. In the hard graph case, if we have an edge weight vector $\mathbf{w}$ with all weighted degrees > 1, achieving $f(\mathbf{w}) = o$, then letting d_{min} be the minimum weighted degree of a vertex, we can set $\mathbf{w}' = \mathbf{w}/d_{min}$ and achieve the lower objective function value $f(\mathbf{w}') = o/d_{min}^2$.

This tendency towards lower degree is not a problem in itself - presumably our learning algorithms can handle uniform scaling of edge weights. However, on many data sets, a large number of degrees in our hard graphs are observed to exactly abut the lower bound of 1. And, the sum of deviations of degrees below 1 must always abut αn in our soft graphs. This implies that our degree restrictions are somehow constraining the natural structure of the graph we are trying to construct - which may be undesirable since the degree restrictions were only originally introduced to avoid degenerate solutions.

One possibility to solve this issue is to return to the objective function originally motivating the one used in [4]:

$$\min_{w_{i,j} \geq 0} \sum_i \|\mathbf{x_i} - \sum_j \frac{w_{i,j}}{d_i} \mathbf{x_j}\|^2$$

This function purely expresses the goal of estimating each vertex as a weighted average of its neighbors. Further, this function avoids the general tendency towards lower degree and so avoids the degenerate $\mathbf{w} = \mathbf{0}$ solution without having to place a constraint on the d_i's. However, it is not convex and it does not avoid the degenerate graphs in which some edges have infinitesimally small weights. These graphs can still be implemented under this objective function by giving some edges infinitely large weight, making the other edges infinitely small in comparison. In order to avoid these cases, we would have to place an upper constraint on degree.

Instead consider the alternative 'non-degree' objective function:

$$\min_{w_{i,j} \geq 0} f(\mathbf{w}) = \sum_i \|\mathbf{x_i} - \sum_j w_{i,j} \mathbf{x_j}\|^2$$

Removing the degree weighting shifts our goal from trying to find a set of edges that best approximates each point as a weighted *average* of its neighbors to trying to approximate each point as a weighted *sum* of its neighbors. This shift allows us to avoid the degenerate graph solutions described above and ensures that a tendency towards low degree does not affect the structure of our

graph. However, as we will see below, this new function, when applied to raw data does not give good results. We will have to use it in combination with complementary data transforms.

First, we note that the non-degree objective function can be minimized as a nonnegative least squares function in the form

$$\min_{\mathbf{w}\geq 0} \|M\mathbf{w} - \mathbf{b}\|^2$$

Let n be the number of data points, d be the number of dimensions in each point, and $e = \frac{n(n-1)}{2}$ be the number of possible edges in a graph over our points. $\mathbf{w}$ is the $e \times 1$ vector of edge weights. $\mathbf{b}$ is the $(n * d) \times 1$ vector created by stacking our data vectors. If each data vector $\mathbf{x_i}$ is $d \times 1$ then $\mathbf{b^T} = [\mathbf{x_1}^T\ \mathbf{x_2}^T\ ...\ \mathbf{x_n}^T]$. Finally, M is an $(n * d) \times e$ matrix. For each $\mathbf{x_i}$, let M_i be the $d \times e$ matrix whose k^{th} column is $\mathbf{0}$ if $i \notin e_k$ and is $\mathbf{x_j}$ if $i \in e_k$ and $e_k = (i, j)$. $M^T = [M_1^T\ M_2^T ... M_n^T]$.

It is not hard to see that $M_i\mathbf{w} = \sum_j w_{i,j}x_j$. So $M\mathbf{w} - \mathbf{b} = [(-x_1 + \sum_j w_{1,j}x_j)^T ... (-x_n + \sum_j w_{n,j}x_j)^T]^T$. So, the magnitude of $M\mathbf{w} - \mathbf{b}$ gives us the value of our objective function.

My implementation of the non-degree optimization function is very similar in structure to the implementation of soft graph optimization program in [4]. We have:

$$\frac{df}{d\mathbf{w}} = 2M^T(M\mathbf{w} - \mathbf{b})$$

At the minimum solution, we will have $\frac{df}{d\mathbf{w}} = \mathbf{0}$. If we minimize over a subset of edges, then in our solution we will also have $\frac{df}{d\mathbf{w}}(i) = 0$ for all edges e_i included in our subset. However, we may have negative derivative on some edges that we excluded. The negative derivative means that we can improve our objective function by increasing the weights on these excluded edges. So, we initially solve our optimization problem over a small subset of edges. At each step, we add more edges to the subset that we solve over, adding those edges with minimal derivatives first. Once all excluded edges have 0 derivative, then we know that $\frac{df}{d\mathbf{w}} = \mathbf{0}$ and we have reached an optimal solution.

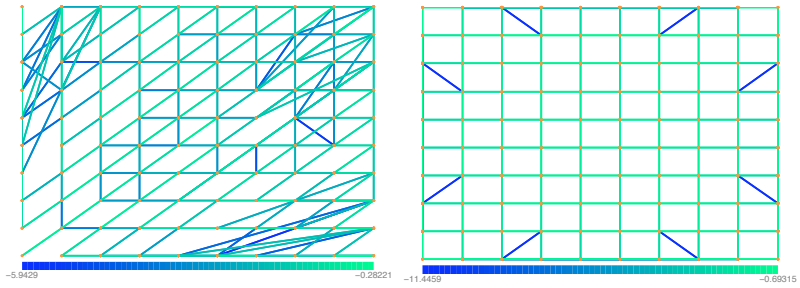
As will be shown below, solving the non-degree optimization problem can be much faster than solving the soft and hard graph optimizations from [4]. My code is rudimentary, and further speed ups are most likely possible through better memory allocation in the code and by more intelligently choosing in what order and at what rate edges are added to the graph.

2.2 Combination with data transformations

Applied to raw data, the above objective function gives odd behavior due to differing data vector magnitudes. Since we do not weight by vertex degree, in the optimal graphs we tend to get datapoints with lower magnitudes connected by low weight sometimes long range edges to datapoints with higher magnitudes. The lower magnitude datapoints can be very closely reconstructed by summing over these edges. At the same time, the low weight edges have little effect on the estimates of the higher magnitude vectors. Without degree weighting, it is possible for the vector $(1, 1)$ to be reconstructed exactly with a single low weight edge to the vector $(100, 100)$. With degree weighting, we could only reconstruct $(1, 1)$ exactly as a weighted average of some points with x values ≤ 1, some points with x values ≥ 1, and some points with y values on both sides of 1. It is easy to see the difference in these two methods by looking at the results over a 10×10 two dimensional grid.

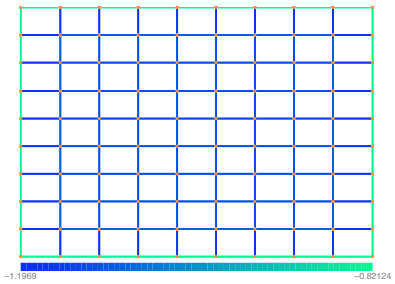
Figure 3: Results over 10×10 2-dimensional grid

(a) Non-degree Objective Function (b) Degree weighted hard graph



One method to improve the behavior of the non-degree graph is to transform all data vectors to have equivalent magnitude. Since the solution to the optimization is invariant to a uniform scaling of the data vectors, we can transform so that specifically $\|x_i'\| = 1 \; \forall \; i$. With all vectors at norm 1, in order to accurately reconstruct the vertices from their neighbors, the graph must have each weighted degree at least somewhat close to one. The odd interplay between high and low magnitude vectors will tend to disappear. A particularly striking illustration of this technique is seen with the results of the non-degree optimization on the 10×10 2-dimensional grid after we apply a Gaussian kernel transform (described below). This transform produces a transformed set of input vectors all with magnitude 1.

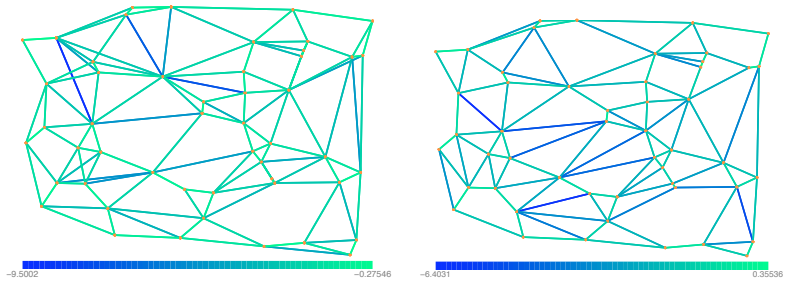
Figure 4: Non-degree graph over Gaussian tranformed 10×10 grid.



The graph in fact perfectly follows the lines of the grid. The non-degree function in correspondence with the same Gaussian kernel transform, when applied to a random set of 2-dimensional data points also gives at least visually similar results to the original hard graph optimization.

Figure 5: Results over 50 randomly placed 2-dimensional points

(a) Non-degree w/ Gaussian transform (b) Degree weighted hard graph



I tested three different methods for transforming our data points to have uniform magnitude 1. Each attempts to at least somewhat preserve the relationships between the original data points.

2.2.1 Projection onto the unit sphere

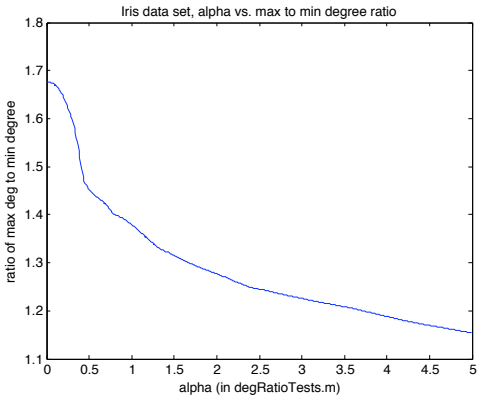
Aside from naively scaling each data vector to have magnitude 1 (a transform which does not adequately preserve the relationships between the vectors), one obvious way to transform the data to have magnitude 1 is to use stereographic projection to project the data on a sphere. Specifically, with d-dimensional data, we can project the data onto a $(d+1)$-dimensional sphere. One important parameter to choose for this projection is the radius of the sphere. (Or, if the radius is set to a constant 1, how much we uniformly scale down our data vectors before applying the projection). The ratio between the radius of the sphere and the spread of our data determines how 'flat' our projection is. If the radius is small, our data may be projected onto most of the surface of the sphere. This can cause odd side effects because points far from the data center will tend to be projected to similar locations near one pole of the sphere. If the radius is large, the data is only projected onto a small portion of the sphere's surface. So, the distances between the projected points closely align with the distances between the original points.

It is worth noting that I also tried to recenter the points to have center of mass at $\mathbf{0}$ and then to apply a method outlined in [7] to project the data onto a sphere such that the center of mass of the projected points was also at $\mathbf{0}$. However, I did not get significantly different results using this technique.

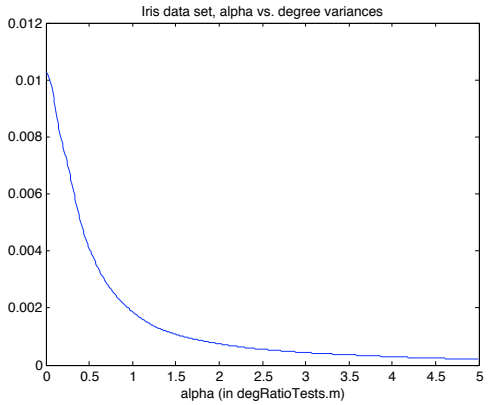
2.2.2 Lifting of points onto plane

The possibility of using a large radius for the stereographic projection of the data points gave rise to the idea of simply lifting the data points onto a $d+1$ dimensional plane by adding a new dimension to each vector that is set to a constant value. If the added dimension has a large magnitude in comparison to the data spread, this transformation approximates projecting the data onto a large radius sphere.

This transformation also allows us to more clearly see how our projection is affecting the resulting graphs. The added dimension, with a constant value across all our data points, gives preference to graphs where each vertex has degree near 1. If each data point $\mathbf{x_i}$ is given value α in the added dimension, the predicted datapoint $\hat{\mathbf{x}}_\mathbf{i}$ will have value αd_i in the added dimension. So, over the whole graph, our added dimension will lead to a cost of $\sum_i(\alpha - \alpha d_i)^2 = \sqrt{\alpha}\sum_i(1-d_i)^2$. The higher α, the more important restricting each d_i to be near 1 becomes. To demonstrate this, 500 runs of the optimization with values of α ranging from 0 to 5 were performed on the IRIS data set, consisting of 100 4-dimensional data points [6]. There was a clear negative relationship between α and the ratio between the maximum and minimum degrees and the variance of degrees in the graph.



(c) max degree / min degree



(d) Degree variance

Similar results were seen when performing the same test over a sets random data points. The above analysis 'exposes' the lifting to plane data transform (and by extension, the spherical transform) as seemingly doing something very similar to the soft graph, except with the possibly undesirable properties that we penalize any deviation of degree from 1, even if it is positive. If, in order to achieve reasonable results using the non-degree method, we must still restrict degrees to be near 1, it may be that we are not accomplishing our original goal of achieving more 'natural' graphs less shaped by degree constraints.

2.2.3 Gaussian kernel transform

The final transform I consider, the Gaussian kernel transform, while possibly not the most practical, proved to be the most interesting. The basic idea is to calculate a new distance between each pair of vectors based on the Gaussian kernel. This distance corresponds to the probability density function at one vector under a Gaussian distribution centered at the other vector and is given by:

$$K(\mathbf{x_i}, \mathbf{x_j}) = e^{-\|\mathbf{x_i}-\mathbf{x_j}\|^2/2\sigma^2}$$

where σ is a chosen parameter. I simply chose $\sigma = 1$ in all my tests. Note that $K(\mathbf{x}, \mathbf{y}) \geq 0$ for all $\mathbf{x}, \mathbf{y}$, and $K(\mathbf{x}, \mathbf{x}) = 1$.

We construct the gram matrix G, where $G_{i,j} = K(\mathbf{x_i}, \mathbf{x_j})$. It is well known that G is positive-semidefinite. This fact can be shown by first using a Taylor expansion to show that $K(i, j)$ can be calculated as the inner product of two infinite dimensional vectors $\phi(\mathbf{x_i})$ and $\phi(\mathbf{x_j})$, and then by applying Mercer's theorem [8]. Since G is positive semidefinite, we can use a Cholesky decomposition to find a lower triangular $n \times n$ matrix L such that $LL^T = G$. The rows of L correspond to n new n-dimensional data points $\hat{\mathbf{x_1}}, \hat{\mathbf{x_2}}...\hat{\mathbf{x_n}}$ where $\hat{\mathbf{x_i}}\hat{\mathbf{x_j}}^T = K(\mathbf{x_i}, \mathbf{x_j})$ for all i, j. This means that $\hat{\mathbf{x_i}}\hat{\mathbf{x_i}}^T = \|\hat{\mathbf{x_i}}\|^2 = K(\mathbf{x_i}, \mathbf{x_i}) = 1$ for all i. So each of our transformed data vectors has magnitude 1.

It is also possible to compute the transformed vectors by using an eigendecomposition of G which gives us a positive diagonal eigenvalue matrix Λ and an orthogonal eigenvector matrix Q such that $Q\Lambda Q^T = G$. Our new data vectors are then given by the rows of $Q\Lambda^{-1/2}$. In some of my code, I use this alternative technique, because for unknown computational reasons, the Matlab implementation of the Cholesky decomposition fails on some large matrices.

One interesting question that I explored but did not succeed in answering is whether it is possible to perform the non-degree optimization over our points using the Gaussian kernel transform without actually explicitly calculating out our transformed vectors. If this were possible, we could possibly avoid the added cost of the Cholesky (or Eigen) decomposition and the costs of optimizing over n rather d dimensions.

I initially tried tackle this problem by looking at the dual of the non-degree optimization problem. We can define the primal function:

$$\theta_p(\mathbf{w}) = \max_{\boldsymbol{\alpha} \geq 0} \|M\mathbf{w} - \mathbf{b}\|^2 - \boldsymbol{\alpha}^T\mathbf{w}$$
$$\theta_p(\mathbf{w}) = \max_{\boldsymbol{\alpha} \geq 0} \Lambda(\mathbf{w}, \boldsymbol{\alpha})$$

where $\Lambda(\mathbf{w}, \boldsymbol{\alpha}) = \|M\mathbf{w} - \mathbf{b}\|^2 - \boldsymbol{\alpha}^T\mathbf{w}$. Since having any element of $\mathbf{w}$ less than 0 will allow us to choose α to drive θ_p to infinity, we know that $\min_{\mathbf{w}} \theta_p(\mathbf{w}) = \min_{\mathbf{w} \geq 0} \|M\mathbf{w} - \mathbf{b}\|^2$.

We define the analogous dual function:

$$\theta_d(\boldsymbol{\alpha}) = \min_{\mathbf{w}} \Lambda(\mathbf{w}, \boldsymbol{\alpha})$$

And, we know that $\max_{\alpha \geq 0} \theta_d(\boldsymbol{\alpha}) = \min_{w \geq 0} \|M\mathbf{w} - \mathbf{b}\|^2$ because our function is convex. We can solve $\frac{d\Lambda}{d\mathbf{w}} = 2M^TM\mathbf{w} - 2M^T\mathbf{b} - \boldsymbol{\alpha}$. Setting the derivative to 0 and plugging into θ_d, we get:

$$\theta_d(\boldsymbol{\alpha}) = -1/2[\boldsymbol{\alpha}^TT\boldsymbol{\alpha} + 4\mathbf{b}^TMT\boldsymbol{\alpha} + 2\mathbf{b}^TMT2M^T\mathbf{b}]$$

Where $T = 1/2(M^T M)^{-1}$. And so, maximizing over $\boldsymbol{\alpha} \geq 0$ we get:

$$\max_{\boldsymbol{\alpha} \geq 0} \theta_d(\boldsymbol{\alpha}) = \max_{\boldsymbol{\alpha} \geq 0} -1/2[\boldsymbol{\alpha}^T T \boldsymbol{\alpha} + 4\mathbf{b}^T MT\boldsymbol{\alpha} + 2\mathbf{b}^T MT2M^T\mathbf{b}]$$

$$\max_{\boldsymbol{\alpha} \geq 0} \theta_d(\boldsymbol{\alpha}) = \min_{\boldsymbol{\alpha} \geq 0} \boldsymbol{\alpha}^T T \boldsymbol{\alpha} + 4\mathbf{b}^T MT\boldsymbol{\alpha}$$

This is our dual problem, and can be solved using quadratic programming. Further $M^T M = \sum_i M_i^T M_i$. Each row of M_i^T is either 0 or a data vector (n-dimensional as a result of the Gaussian transform). So $M_i^T M_i(i,j) = \mathbf{m_i}^T \mathbf{m_j}$ where $\mathbf{m_i}, \mathbf{m_j}$ are the i^{th} and j^{th} rows of M_i^T respectively. This means that either $M_i^T M_i(i,j) = 0$ or $M_i^T M_i(i,j) = K(\mathbf{x}, \mathbf{x}')$ for some data vectors $\mathbf{x}$ and $\mathbf{x}'$ in our original data set. We can show similarly that the elements of $\mathbf{b}^T M$ can be calculated by simply calculating the Gaussian kernel over pairs of our original data vectors. Since T can be calculated from $M^T M$, this shows that we can solve the above dual problem to find the non-degree graph without ever actually expanding out the Gaussian transformed vectors.

Of course, the elephant in the room is the computation of T and the fact that T is $n \times n$. Avoiding the inverse computation and in general formulating the optimization to avoid the slow down that comes from the blow up of dimension to n is still open for work. It is worthwhile to note that, as described below, our solutions to the optimization over the Gaussian transformed data seem experimentally to be extremely sparse - with a similar number of edges to solutions of the optimization over the original d dimensional data. So the runtime increase from blowing up to n dimensions is not catastrophic.

The Gaussian kernel method seems to give similar results to the spherical projection and lifting to plane transformations. All these transformations seems to give pretty similar results to the soft and hard graphs produced by optimizing the original objective function. This fact is very interesting, because, while the other projections maintain degree at $d+1$, the Gaussian kernel transform leads to a dimensional blow up from d to n. At the same time, the graphs produced by optimizing over the Gaussian transformed points have similar average (non-weighted) degree to the graphs produced by optimizing over the $d+1$ dimension transformed points.

In [4], the authors show an upper bound on average degree of $2(d+1)$ for the hard and soft graphs. An analogous upper bound of $2d$ holds for the non-degree graphs. We minimize $f(\mathbf{w}) = \|M\mathbf{w} - \mathbf{b}\|^2$ with $\mathbf{w} \geq 0$. Assume that the sparsest solution to this minimization, $\mathbf{w}$ has $> nd$ edges. Then, since M is an $(n * d) \times e$ matrix, it cannot have rank greater than $n * d$. So, any set of $> nd$ columns of the matrix are linearly dependent. So, there is some $\mathbf{w}'$ with nonzero weights only on edges with non zero weights in $\mathbf{w}$ such that $M\mathbf{w}' = 0$ (since by our assumption there are $> nd$ of these edges). Using the same idea from [4], we can scale down $\mathbf{w}'$ until adding (or subtracting) it from $\mathbf{w}$ leaves all edges with non zero weights but causes at least one positive weight to go to 0. This gives us an optimal graph with total number of edges 1 less than our assumed sparsest solution, giving a contradiction. So, our sparsest solution must have $\leq nd$ edges - which corresponds to $\leq 2d$ average degree.

When we apply the Gaussian transform and increase dimensionality from d to n, we also also blow up in this bound to $2n$ (which is in fact higher than the average degree of the complete graph). However, since we do not see a corresponding increase in average degree, it seems that the conjecture that the average degree of the graphs somehow corresponds to the underlying dimension of the data may be coming into play. Note: the authors' conjecture was of course about the hard and soft graphs of [4]. Although my initial observations were that the average degree of the non-degree graph did not blow up when using the Gaussian transform, if we apply the Gaussian transform and then calculate hard and soft graphs, we also do not see a significant increase in average degree.

I explore the relationship between underlying dimension and average degree more in Section 3, as it seems to be an important property of our graphs.

2.3 Results using non-degree function in two-dimensions

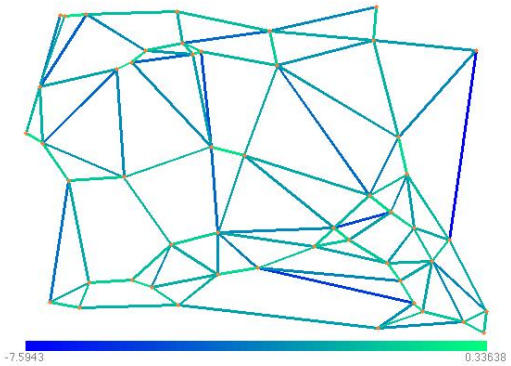
While the non-degree function with the data transforms give very similar results to the hard and soft graphs from [4], there are a few noticeable differences. For a given two dimensional data set,

there always exists a planar hard and soft graph - and because of the technique we use of iteratively building up edges, we almost always get this planar graph as a result of our optimization. The non-degree graphs are also planar to some extent, however, in nearly every two dimensional data set, we see at least some crossing edges.

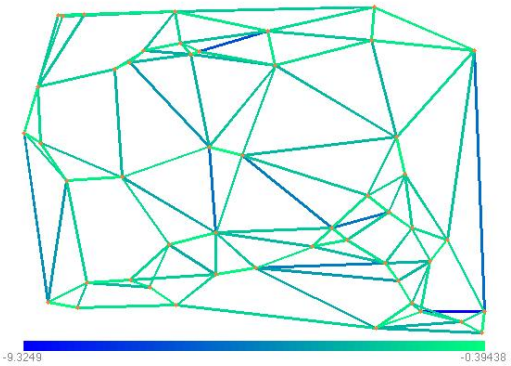
In the graphs displayed below, we can see that with a larger number of data points, a larger number of long range connections begin to emerge in the lifted plane and spherical projection graphs. This is not ideal, and it seems that the Gaussian transform may give better performance with large data sets. However, it is worth noting that the long range connections typically have low weights as indicated by the deeper blue colors in the graphs below. Note: the scale at the bottom of the graphs maps is by $log(w_{ij})$.

Figure 6: Results over 50×2 random data matrix

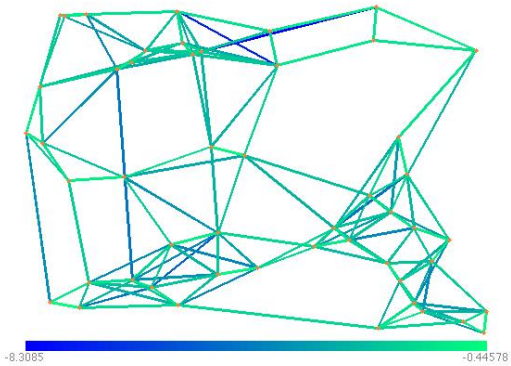
(a) .1 soft graph



(b) Non-deg w/ Gaussian transform



(c) Non-deg w/ planar lift, $\alpha = 5$



(d) Non-deg w/ sphere projection, $r = 2$

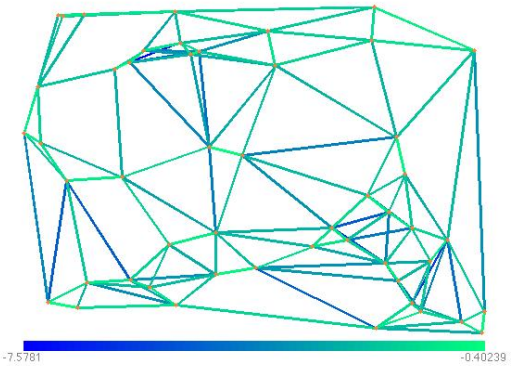
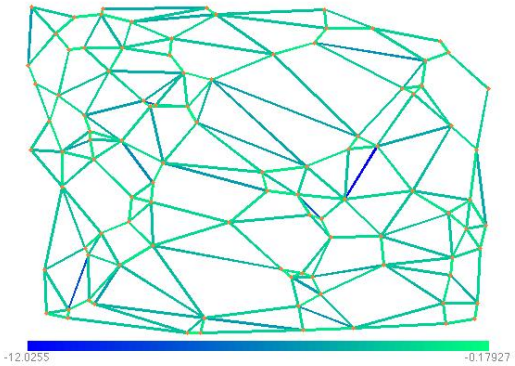
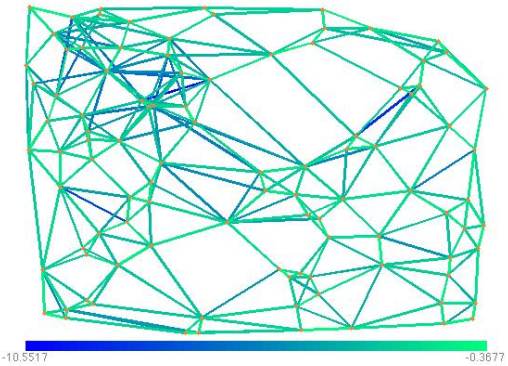


Figure 7: Results over 100×2 random data matrix

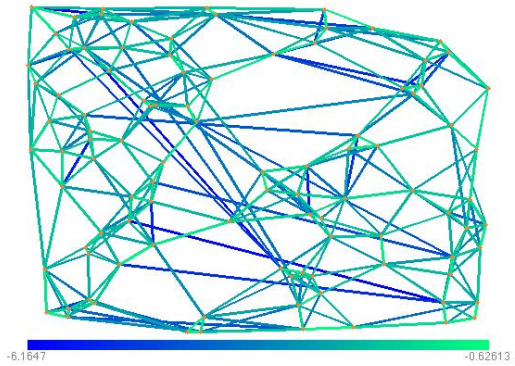
(a) .1 soft graph



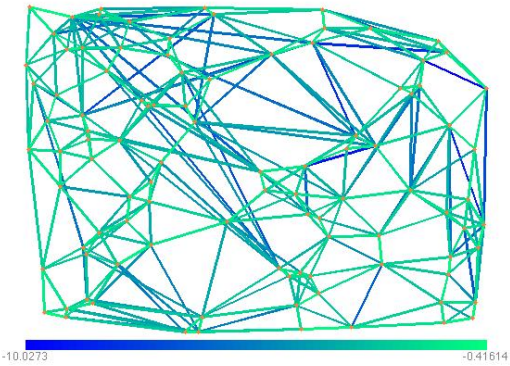
(b) Non-deg w/ Gaussian transform



(c) Non-deg w/ planar lift, $\alpha = 5$



(d) Non-deg w/ sphere projection, $r = 2$



2.4 Learning Results

The following table includes some values of classification and regression accuracy, runtime, and average graph degree for the non-degree and soft graph techniques. Data sets are taken either from the UCI repository [6], or the LIBSVM repository [2]. In these tests, we normalize each data column to have variance 1, and we measure classification accuracy over 100 runs of removing a random 10% of the data points and running label propagation using the remaining 90% of the data points as our labeled examples. In Abalone, because there are so many data points and because we compare directly to results from [4], we use 2-fold, rather than 10-fold cross validation. We measure classification error as a percent and regression error as mean squared error, where the labeled values are scaled to have variance 1. We use $\alpha = 0.1$ for the soft graph, and set the radius of our sphere and height of our plane to the maximum value in our data matrix for the sphere and plane transforms. Runtime improvements for the non-degree sphere and plane graphs are significant and are largely the result of not having to do a search over μ to achieve the desired α in the soft graph. Although accuracy is nearly uniformly worse with these techniques, on large data sets with thousands of points they may preferable to the soft graph technique due to decreased runtime. Run times for construction of the graph over the Gaussian transformed data are significantly slower due to the large M matrix that must be constructed at each iteration of the optimization. Accuracy with the Gaussian transform occasionally beats the soft graph, however is in general inconsistent. So, it seems that as a practical machine learning technique, the non-degree graph with the Gaussian transform may not be viable.

Table 1: Learning results for difference graph types

Data Set	Source	Type	n	dim	Soft Graph			Non-degree Gauss			Non-degree Plane			Non-degree Sphere		
					time (sec)	error	avg deg	time (sec)	error	avg deg	time (sec)	error	avg deg	time (sec)	error	avg deg
PIMA	UCI	2 classes	768	8	222.0	27.92	10.69	989.7	**26.0**	18.27	25.6	27.13	8.49	44.4	27.36	10.12
SONAR	UCI	2 classes	208	60	22.7	**8.40**	13.04	3.01	23.65	5.34	3.71	9.95	13.43	4.03	9.40	13.37
IRIS	UCI	3 classes	150	4	2.84	**3.53**	6.80	4.79	5.13	7.48	.86	8.27	5.01	.71	6.27	5.26
VEHICLE	UCI	4 classes	846	18	74.65	**22.54**	11.45	1186.6	28.1	17.25	6.60	31.24	3.88	6.73	31.43	5.07
IONOSPHERE	UCI	2 classes	351	34	87.13	**5.46**	11.80	43.98	25.66	10.40	10.41	4.60	14.21	15.78	5.94	16.31
GLASS *	UCI	6 classes	214	9	10.77	26.90	8.11	18.02	**24.33**	8.52	1.04	32.86	4.27	1.08	33.86	4.47
HEART	UCI	2 classes	270	13	7.38	**18.78**	11.022	45.00	22.26	29.09	1.15	20.44	8.18	1.68	19.85	8.67
ABALONE **	LIBSVM	regression	4177	8	1582	**0.479**	12.7	NA	NA	NA	95.81	0.743	7.20	72.60	0.586	7.51
HOUSING	UCI	regression	506	13	23.84	**0.128**	10.14	121.60	.163	12.28	3.11	0.195	5.06	2.39	0.147	5.29
MACHINE	UCI	regression	209	6	5.95	0.179	7.92	13.37	0.334	8.97	2.04	0.162	5.92	3.01	**0.150**	5.86

* When normalizing the variance of each dimension of GLASS to 1, we greatly magnify the first and fifth dimensions. So we do not normalize in this case.
** Soft graph data taken from [4] to avoid long run time.

2.5 Further questions

My work with the alternative non-degree objective function gave rise to many questions that still need to be explored. Most importantly, the original motivation for moving to the non-degree function was that it seemed somehow unnatural that in the hard graphs we were producing we were seeing so many degrees abutting the lower bound of 1. The non-degree graph avoided having to place degree restrictions on the graph. However, in order to get reasonable results, we had to transform our data in a manner that essentially reintroduced degree constraints. It could be interesting to more formally pin down how much a graph structure is being determined by the objective of reconstructing points as weighted averages (or sums) of their neighbors, and how much the structure is being constrained by degree or other requirements. It may also be possible to find some other formulation that avoids the use of degree constraints entirely.

Either way, the non-degree technique may prove useful because the optimization problem is somewhat simpler than the original hard or soft graph optimization problems. The M matrix is easy to understand and does not have an additional d rows used to implement degree constraints. So, the analysis of its rank and other properties may be easier. In addition, the non-degree problem essentially builds our degree constraints into the data transform that we use. It may be possible to choose data transformations that lead to effective degree constraints that are somehow adapted to the actual structure of our data, rather than predetermined.

3 Relation of graph average degree and data dimensionality

The above observations of relatively low degree graphs built with the Gaussian transform helped motivate some exploration into the relationship between underlying data dimensionality and average degree in the constructed graph. In [4], the authors conjecture that the average degree of the graphs is somehow related to the 'natural' dimension of the data. As explained above and in [4], the number of edges in our optimal graph is upper bounded by the rank of $\begin{bmatrix} M \\ A \end{bmatrix}$, when calculating the hard and soft graphs (and by the rank of M for non-degree graphs). For the hard and soft graphs, this upper bound is $n(d+1)$. To keep things simple, in the explanations below, I assume that we are working with the soft graph.

We consider the simplest example we can of having higher dimensional data with a low underlying dimension - when some dimensions are simply linear combinations of others. If we have an $n \times d'$ data matrix, with rank $d < d'$ (i.e. our columns are all just linear combinations of d basis vectors), then we could expect our graph over the d' dimensional data to be sparser than a typical graph over d' dimensional data with independent columns of the data matrix. Specifically, in the soft graph optimization, M is a $d'n \times e$ matrix, made up of d' $n \times e$ matrices stacked on top of each other. Each of these matrices, which is $(Y^{(i)}U^T)^T$ in [4], and which I call M_i, has columns indexed by edges and rows indexed by vertices. The column corresponding to the edge between x_k and x_j, $e_{k,j}$ has zero at all elements except for the k^{th} and j^{th} elements, where it has the difference between the two data vectors in the i^{th} dimension. Writing the matrix out:

$$M_i = \begin{bmatrix} x_1(i)-x_2(i) & x_1(i)-x_3(i) & \dots & x_1(i)-x_n(i) & 0 & \dots & 0 & 0 \dots & 0 \\ x_2(i)-x_1(i) & 0 & \dots & 0 & x_2(i)-x_3(i) & \dots & x_2(i)-x_n(i) & 0 \dots & 0 \\ 0 & x_3(i)-x_1(i) & \dots & 0 & x_3(i)-x_2(i) & \dots & 0 & 0 \dots & 0 \\ \vdots & \vdots & & \vdots & \vdots & & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & x_n(i)-x_1(i) & 0 & \dots & x_n(i)-x_2(i) & 0 \dots & x_n(i)-x_{n-1}(i) \end{bmatrix}$$

If we look at all d rows of M corresponding to a single vertex v, we get the submatrix M_v of M which looks like:

3

Each row is identical except for the dimension that its values are from. So, it is clear that if the columns in our data matrix can be spanned by d basis vectors, than so can the rows of M_v. So, the

rank of M_v is at most d, so the rank of M in general is at most nd, and so the rank of $\begin{bmatrix} M \\ A \end{bmatrix}$ is at most $n(d+1)$, giving an upper bound on the number of vertices in our graph (and an upper bound of $2(d+1)$ for average degree).

This upper bound aligns with our intuition that if the underlying dimensionality of our d' dimensional data is actually $d < d'$, then we will see a sparser graph than if the underlying dimension were actually d'. However, the above result only applies to the *upper bound* on degree. In reality, our graphs rarely reach this upper bound. Experimental results confirm that even when not reaching the upper degree bound, graphs over data sets with extra dimensions that are linear combinations of a smaller set of dimensions tend to have approximately the same degree as the graphs constructed over the lower dimension underlying data. I constructed soft graphs over 10 random matrices for each even dimension from 2 through 50. For each data matrix, I constructed a corresponding data matrix with 4 times the number of dimensions, where each additional dimension column was a randomly generated linear combination of the original columns. I then took the ratio between the average degree of the lower dimensional graph and the average degree of the higher dimensional graph. Over the 250 graphs, this ratio was 1.0263 on average (so actually a slightly higher average degree on the lower dimensional graphs). The maximum ratio value observed was 1.1532 and the minimum was 0.9114. So, even beyond the upper degree bound, we are seeing an extremely close correspondence between the degree of the higher dimensional graph and the degree of the corresponding low dimensional graph. Note: This similarity between average degree does not imply a similarity between the actual graphs. By changing the relative weights of different dimensions we may actually significantly change the connections present in our graph. Points that are relatively close to each other in the low dimensional data may be very far from each other in the high dimensional data.

Similar results to those above are seen when the higher dimensional data vectors are not simply linear functions of the low dimensional data but are, for example, polynomial functions, or the Gaussian kernel transform function. This is somewhat more surprising because we cannot even easily show that the upper bound on degree remains the same after a nonlinear transform.

We can construct examples where the above results fail. For example, in one test, I generated a random $n \times 10$ data matrix, X where the values in the first dimension were chosen uniformly on the interval $[0, 100]$, and the values in all other dimensions were chosen uniformly on the interval $[0, 1]$. I then generated a series of linearly transformed data matrices Y with ≥ 10 dimensions. Each of these new data matrices used weights that equalized the disparity in average magnitude between the first dimension and the other dimensions. The simplest such Y is the $n \times 10$ matrix equal to X, but with each column normalized to average value 1. Calculating the soft graph over this Y for $n = 50$ gives average degree around 8.2, vs. an average degree of around 5.6 for X (exact values depending on the randomly generated data points.)

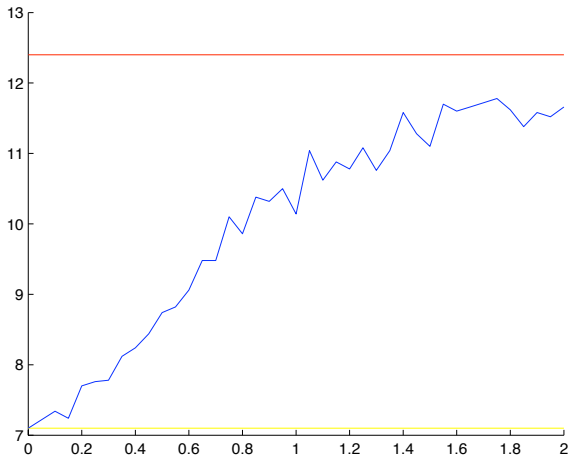
Intuitively, this blow up in degree seems to be because the graph over X had lower than usual degree in the first place because the data in X was *approximately* 1 dimensional - with a large amount of the variation between vectors arising from the variation in the first, higher magnitude, dimension. By equalizing the average magnitude of the dimensions, we pulled the points away from the line that they approximately fell on, removing the approximate 1-dimensionality and causing a corresponding increase in average degree.

The fact that approximate low dimensionality can lead to low degree graphs is very important because for real world data sets a set of high dimensional data points will never lie exactly on a low dimensional plane or manifold.

One representative test of how lying approximately on a low dimensional plane or manifold affects average degree was obtained by generating a 100×5 data matrix X with each element chosen uniformly between 0 and 1. A new matrix Y with 25 dimensions, all linear combinations of the initial 5 dimensions was then generated. Y was then randomly perturbed at each position by a value choose uniformly in the range $[0, \delta]$, where δ was varied between 0 and 2. For each perturbation, the average degree of the soft graph calculated over the points was plotted. The yellow line at the bottom of the graph indicates the average degree of the soft graph calculated over the unperturbed Y. The red

line at the top indicates the average degree of the soft graph over a randomly generated 100×25 data matrix. It is interesting that even with $\delta = 2$, which is twice of the magnitude of even the largest values in our original matrix, we still see a significant degree reduction.

Figure 8: Variation of average soft graph degree with deviation from low dimensional plane.



Unfortunately, aside from experimental results, we do not currently have a great understanding of how lying on, or approximately on, a low dimensional manifold causes a reduction in the degree of our graphs. Even for the simplest case - in which the high dimensional data lies exactly on a low dimensional plane, although we have an explanation of why our graph must obey a reduced upper bound on average degree, we cannot yet explain why the dimension stays nearly equal to the dimension of the graph constructed over the underlying low dimensional data.

4 Learning over the graphs

While the original 'Fitting a Graph to Vector Data' paper and much of my work on this project focused on exploring the properties of the graphs produced by the hard, soft, and non-degree objective functions, there is also a lot of work to be done in understanding exactly why these graphs are useful in learning problems. I discuss a couple of issues with learning over the graphs below.

4.1 Relationship between Laplacian based learning techniques and Laplacian squared graph construction

One original motivation given in [4] was for the soft and hard graphs to serve as general inputs for graph based learning algorithms - including clustering algorithms, regression algorithms, and classification algorithms. In [4], regression and classification was performed by using the label propagation algorithm described in [14] and [15]. Clustering was performed using a standard spectral clustering algorithm described in [10], [13], and others.

The idea of the label propagation algorithm described in [14] is to predict a vector $\mathbf{f}$ of function values over our points that minimizes an energy function over the graph:

$$E(\mathbf{f}) = \frac{1}{2} \sum_{i,j} w_{ij} (\mathbf{f}(i) - \mathbf{f}(j))^2$$

$$E(\mathbf{f}) = \mathbf{f}^T \Delta \mathbf{f}$$

where Δ is the graph Laplacian. This minimization is subject to $\mathbf{f}$ matching the labeled values of our function. The energy function gives the 'smoothness' of our function over the graph - if the

function changes significantly over highly weighted edges, the energy will be high. So, minimizing the energy function causes us to predict similar values of f on strongly connected vertices or groups of vertices.

If we perform an Eigendecomposition of the graph Laplacian, we get $\Delta = \phi\Lambda\phi^T$, since the Laplacian is positive semidefinite. Expanding into a sum we get: $\Delta = \sum_i \phi_i\lambda_i\phi_i^T$. So we can rewrite our energy function:

$$E(\mathbf{f}) = \mathbf{f}^T\Delta\mathbf{f}$$
$$E(\mathbf{f}) = \sum_i \lambda_i\mathbf{f}^T\phi_i\phi_i^T\mathbf{f}$$
$$E(\mathbf{f}) = \sum_i \lambda_i(\mathbf{f}\cdot\phi_i)^2$$

And, $\phi_i^T\Lambda\phi_i = \phi_i^T(\sum_j \phi_j\lambda_j\phi_j^T)\phi_i = \phi_i^T\phi_i\lambda_i\phi_i^T\phi_i = \lambda_i$ since our eigenvectors are orthonormal. So λ_i gives the smoothness of the i^{th} eigenvector over the graph. So, minimizing the transformed energy function above can be thought of as maximizing how similar $\mathbf{f}$ is to the eigenvectors of the graph, with higher weights placed on smoother eigenvectors.

In spectral clustering, the most basic idea is to take a certain number of the smoothest eigenvectors of the graph (those with the lowest eigenvalues), and to cluster our points based not on their original values, but on their values in these eigenvectors. (Note: each eigenvector has n elements, one corresponding to each of our data points/graph vertices). In order for these eigenvectors to be smooth, strongly connected vertices and sets of vertices in the graph must have similar values on these eigenvectors. So clustering by the values on the smoothest eigenvectors will tend to place vertices that are part of strongly connected subgraphs into the same clusters.

The moral of the rambling story given above is that the regression, classification, and clustering algorithms applied to the graphs built in [4] are all based on maximizing smoothness over the graph - or in other words, on minimizing the energy function given by $\mathbf{f}^T\Delta\mathbf{f}$. Or, if each element of f is a vector so f is an $n \times d$ matrix, with rows $\mathbf{f_1}...\mathbf{f_n}$ and columns $\mathbf{c_1}...\mathbf{c_d}$, then these methods seek to minimize

$$E(f) = \frac{1}{2}\sum_{i,j} w_{ij}\|\mathbf{f_i} - \mathbf{f_j}\|^2$$
$$E(f) = \sum_{k=1}^{d}\frac{1}{2}\sum_{i,j} w_{ij}(\mathbf{f_i}(k) - \mathbf{f_j}(k))^2$$
$$E(f) = \sum_{k=1}^{d}\mathbf{c_k}^T\Delta\mathbf{c_k}$$
$$E(f) = \sum_{k=1}^{d}(f^T\Delta f)_{k,k}$$

that is, the energy is the sum over the diagonal elements of $f^T\Delta f$.

The actual graph construction process seeks instead to minimize $f(\mathbf{w}) = \|\Delta X\|_F^2$ where $\|M\|_F$ is the Frobenius norm $(\sum_{i,j} M_{i,j}^2)^{1/2}$. The square of this norm is also given by summing the diagonal

elements of $M^T M$, so in our case,

$$f(\mathbf{w}) = \sum_{k=1}^{d} [(\Delta X)^T (\Delta X)]_{k,k}$$

$$f(\mathbf{w}) = \sum_{k=1}^{d} [X^T \Delta^T \Delta X)]_{k,k}$$

$$f(\mathbf{w}) = \sum_{k=1}^{d} [X^T \Delta^2 X)]_{k,k}$$

In the single dimensional case, we simply have $f(\mathbf{w}) = X^T \Delta^2 X$. Or, decomposing Δ^2 into its eigenvectors, which are the same as the eigenvectors of Δ except with squared eigenvalues, we have: $f(\mathbf{w}) = \sum_i \lambda_i^2 (X^T \cdot \phi_i)^2$.

From these manipulations, we see that the objective function optimized to build our graph is exactly the same as the objective function optimized when learning over our graph, except that graph construction uses Δ^2, and graph learning uses Δ. When decomposing into eigenvectors, we can see that the only difference is that we weight by squared eigenvalues in the graph construction phase.

It is may be worth exploring the connection between the similar optimization problems used in the complementary graph construction and learning phases. Does it make sense to use the Laplacian squared when building a graph for the above learning techniques. If so, why? It may be possible to find good (and possibly very sparse) approximations for the graphs by looking for graphs that well approximate the laplacian squared. These approximate graphs would have approximately optimal solutions of the objective function. However, does it make more sense to look for approximate graphs that well approximate the laplacian instead? Since, although the squared laplacian determines the objective function value, the laplacian is what will be used when actually applying learning algorithms to the graph.

It is possible that the above questions are closely related to research in nonlinear dimensionality reduction. One popular method of nonlinear dimensionality reduction is locally linear embedding (LLE), described in [12] and [11]. The idea of the LLE algorithm is to take a set of high dimensional vectors and construct a directed graph over these vectors, optimizing the same objective function as used for the hard and soft graphs, except that weighted out degrees are restricted to be exactly 1. A set of lower dimensional vectors is then chosen to minimize the same objective function over the constructed graph, with the constraint that the lower dimensional vectors are orthonormal. In this way, the lower dimensional vectors in some sense are an optimal lower dimensional approximation of the original vectors, preserving the locality relationships present in the original data set.

Diffusion maps are another method used for nonlinear dimensionality reduction (also described in [12]). With diffusion maps, we consider, like with the label propagation technique, minimizing the energy of some function ($n \times 1$ vector) $\mathbf{f}$ over the original high dimensional data - with the constraint that $\|\mathbf{f}\|^2 = 1$. Instead of using a constructed graph, we use a complete graph over the data, with edge weights provided by some distance metric - such as the Gaussian kernel. Of course, the objective function is minimized at 0 when all elements of $\mathbf{f}$ are equal. However, in diffusion maps, the k eigenvectors of Δ with the smallest eigenvalues are considered and used as the columns of a reduced k dimensional data matrix. This is similar to the technique used with spectral clustering and ensures that vectors with similar values on the new lower dimensional data vectors must have had similar values on the original vectors, since the new dimensions correspond to the smoothest eigenvectors.

So, in summary, diffusion maps and the LLE algorithm both attempt to reduce high dimensional data to a lower dimension while preserving locality information. However, like our graph construction, the LLE algorithm is based on an optimization using Δ^2, whereas, like the learning techniques we apply to our graphs, diffusion maps use Δ. There has been some work comparing the results

of these two methods (such as in [1]), with some results coming from the fact that Δ and Δ^2 have the same eigenvectors. Exploring this work more may be helpful in understanding the relationship between the graphs explored for this project, and the machine learning algorithms that are applied to them.

4.2 Learning algorithms designed for use with these graphs

Although the hard and soft graphs were originally proposed as 'general purpose' graphs that could be useful for a variety of learning algorithms, it remains an open question whether we can find effective learning algorithms or techniques specific to these graphs.

One major difficulty with the graph construction process is that it is completely agnostic to dimension - each dimension of our input data is treated equally when constructing the graph. If we have data with many irrelevant dimensions that do not strongly correlate with the function or labels we are trying to predict, our graph may be shaped in part to best approximate these irrelevant dimensions. In the worst case, if the number or weight of irrelevant dimensions overwhelms the number of relevant dimensions, our graph may be almost entirely uninformative about the function we are attempting to predict. In [4], before the graph is constructed, the columns of our data matrix are each scaled to have variance 1. This scaling essentially gives an equal weight to the goal of accurately fitting the graph to our vectors over each dimension. One simple extension that I tested was to instead scale each column in proportion to the covariance of the column with the labels (or function values) over our labeled data points - so accurately predicting data points on dimensions that strongly covaried with the predicted labels would be given greater weight. This modification did not seem to have a significant effect of the data sets that I tested it on. However it may be a worthwhile technique to employ in cases where its seems likely that many irrelevant variables may be present. Alternatively, a traditional supervised machine learning algorithm (such as an SVM) could be run, possibly on a small subset of the data, to help identify and eliminate and/or assign low weight to less relevant dimensions.

An alternative to simply applying a weighting scheme to the dimensions is to actually incorporate labels into the graph construction process - with the hope that these labels will help us build a graph that better informs the value of the function we are trying to predict. However, finding a reasonable way to incorporate labels into graph construction is still an open area to explore. For binary classification, I tested a technique in which an extra dimension was added to the original data. For labeled points this dimension was given the label value - either 0 or 1. For unlabeled points, this dimension was given value 0.5. The graph was then constructed over the modified data, and the label of an unlabeled point could be predicted based on whether or not its reconstructed value on the label dimension was greater or less than 0.5. This technique performed nearly as well as label propagation on the soft graph for the IRIS data set. However, the idea behind the technique seems somewhat misguided. Simply using labeled values as an added dimension does not really provide information about how to treat other dimensions in relation to each other. It does cause labeled points with the same label to be more likely to be connected strongly to each other.

So, one possible modification of this technique could be to construct a graph over the labeled points with the labels included as an added dimension, possibly scaled up so that they have a relatively high importance to the optimization. The deviations of the reconstructed data values from the true values could then be observed. Dimensions with low average deviation could be given higher weight in the full graph construction as they are better predicted when the labels are included in the data, and therefore more likely to vary strongly with the labels. Again however, this method would be focused around properly weighting the dimensions in the graph construction. A cleaner method, in which labels were somehow incorporated into the actual optimization would be ideal. It may even be possible to have a method where different dimensions change in relative importance as we move to different parts of the graph where they become more or less relevant for separation or classification.

5 Uniqueness, planarity, and connection to rigidity of truss structures

In this final section of the paper I will briefly describe work to better understand the optimization problem used to construct the hard and soft graphs and how this problem determines graph properties. This work eventually (if indirectly) led to the discovery of what we believe is a counter example to the conjecture that the hard and soft graphs are unique for data points in general position. I will be writing up this work in more detail in collaboration with Christopher Musco, whom I worked with to find the counter example.

To keep things short for now, I will only be talking about the soft graph optimization function. Specifically, I will be looking at the optimization $\min_{\mathbf{w}} f(\mathbf{w}) + \mu * \eta(\mathbf{w})$ where $w \geq 0$. A solution $\mathbf{w}$ to this equation will also be a solution to the soft graph optimization for a corresponding value of α. And, this optimization problem can be solved by solving the non-negative least squares problem:

$$min_{\mathbf{w},\mathbf{s}\geq 0} \| E \begin{bmatrix} \mathbf{w} \\ \mathbf{s} \end{bmatrix} - \mathbf{b} \|^2$$

Where,

$$E = \left[\begin{array}{c|c} M & 0_{n\times n} \\ \hline \sqrt{\mu} * A & -I_{n\times n} \end{array}\right]$$

and M and A are as defined in [4]. $\mathbf{w}$ is our edge weight vector and $\mathbf{s}$ is a n length vector of slack variables used to multiply by the negative I matrix to bring any degree above 1 down to 1. Finally, b is an $n \times (d+1)$ length vector in the form $\begin{bmatrix} \mathbf{0_{n*d}} \\ \mathbf{1_n} \end{bmatrix}$.

As explained above, the bound of at most $n(d+1)$ edges in the sparsest graph is obtained by arguing that since E has $n(d+1)$ rows it has rank $\leq n(d+1)$. So, if we have $\geq n(d+1)$ edges in our graph, then the columns of E corresponding to those edges are linearly dependent. So, we can find a set of weights on our edges $\mathbf{w}'$ s.t. $\mathbf{w}'$ is 0 on all edges not in our graph and $M\mathbf{w}' = 0$. We can then scale $\mathbf{w}'$ such that when we subtract it from $\mathbf{w}$, no edges get negative weight, however the weight of one edge previously in the graph becomes 0.

We can in fact get a better rank bound than this. We know that $[\sqrt{\mu} * A | -I_n]$ has rank at most n. So $rank(E) = rank(M) + n$. And we can split M up into d matrices $M_1, ...M_d$ - each corresponding to the differences across edge connected vertices in each of the d dimensions. $M_i = UD_i$, where U is defined as in [4] and D_i is the $e \times e$ diagonal matrix with $D_i(a,a) = x_j(i) - x_k(i)$ where $e_a = (j,k)$. Now, we can determine the rank of U using a result from [3]. Each column of U, $\mathbf{u_k}$, corresponds to an edge $e_k = (i,j)$ of our graph. And $\mathbf{u_k}(i) = 1 = -\mathbf{u_k}(j)$. All other elements of $\mathbf{u_k}$ are 0. Now consider any cycle in our graph made up of edges $e_1 e_2 ... e_i$ passing through the vertices $x_1 x_2 ... x_i x_1$. Then looking at the columns of U corresponding to our edges, u_{e_1} (or $-u_{e_1}$) has a weight of 1 in the x_1 position and a weight of -1 in the x_2 position. u_{e_2} (or $-u_{e_2}$) has a weight of 1 in the x_2 position and a weight of -1 in the x_3 position. So, we can add u_{e_1} and u_{e_2} with the appropriate positive or negative signs, to get a vector with a 1 at position x_1, a $(-1+1) = 0$ at position x_2, a -1 at position x_3, and zeros everywhere else. Call this new vector $\hat{u}$. We can add $\hat{u}$ to u_{e_3} or $-u_{e_3}$ to get a vector with a 1 still at postition x_1 but a 0 at position x_3 and a new -1 at postion x_4. Following this pattern around our cycle, some summation $\hat{u} = c_1 * u_{e_1} + ... + c_{i-1} * u_{e_{i-1}}$ will give us a vector with a 1 at position x_1 and a -1 at postion x_i. So, this summation will give us either u_{e_i} or $-u_{e_i}$ since e_i connects x_1 and x_i, completing our cycle. So, the column of U corresponding to e_i is spanned by the other columns of U corresponding to edges in the cycle $e_1 e_2 ... e_i$. In general, any set of columns chosen such that there is a cycle on the graph using the edges corresponding to those columns will be linearly dependent. A set of columns is only linearly independent if its corresponding edge set contains no cycles. The largest set of edges containing no cycles is a spanning tree over our graph, which contains $n-1$ edges. So, U has rank $n-1$. Correspondingly U has a null space of dimension

1. This null space is given by scalings of $\mathbf{1}$ (simply summing the rows of U each with weight 1 yields $\mathbf{0}$).

Now, if M were simply d U matrices stacked on top of one another, then $rank(M)$ would be $n-1$. However, each M_i is a U matrix with a different weighting applied to the columns. We may expect that, in general position, the null spaces of each M_i are disjoint, so, we have M with null space of dimension at least d and therefore rank at most $nd-d$. So we have E with rank at most $nd-d+n = n(d+1)-d$, which is lower than our original bound. However, we can do even better than this bound. In fact, each time we add an M_i matrix to the stack, we loose an extra dimension from our rank. i.e. the null space of $\begin{bmatrix} M_i \\ M_j \end{bmatrix}$ has 3 dimensions. This causes us to have in total $rank(M) = (n-1)+(n-2)+...+(n-d) = \mathbf{n-[d(d+1)/2]}$.

We will explain in the extended write up of this work, that each M corresponds to the rigidity matrix of a truss structure. The null space of M consists of all trivial motions - motions that move the structure but do not deform it. Each trivial motion can shift the structure in any of the d dimensions and/or rotate the structure. With d dimensions, there are $\binom{d}{2}$ possible rotations. So, we have that the total dimension of the null space of the rigidity matix is $d+\binom{d}{2} = d+\frac{d(d-1)}{2} = \frac{d(d+1)}{2}$. So the rank of the rigidity matrix is $n-[d(d+1)/2]$. This is exactly the result given above.

We will see that the analogy to truss structures can extend further, giving us a way to understand when we have reached an optimal graph. Using this analogy will help explain the counter example to uniqueness of the hard and soft graphs over vectors in general position.

6 Conclusion

The work presented in this report covers a wide range of topics relating to the construction of graphs over vector data using the soft, hard, and non-degree objective functions. Throughout the report more questions have been asked than answered about both the properties of the graphs produced and their application to learning problems. While the actual techniques may not prove 'correct' in that they may give way to more natural, efficient, or effective algorithms, further research into why our graphs behave as they do may provide insight into exactly what these better algorithms may be.

References

[1] Belkin, Mikhail and Partha Niyogi. Laplacian Eigenmaps for Dimensionality Reduction and Data Representation. *Neural Computation*, 15: 13731396. 2003.

[2] Chang, C.-C., & Lin, C.-J. (2001). LIBSVM: a library for support vector machines. Software available at http://www.csie.ntu.edu.tw/?cjlin/libsvm.

[3] Chen, Doron. Algebraic and Algorithmic Applications of Support Theory. Tel-Aviv University. 2005.

[4] Daitch, S., Kelner, J., & Spielman, D. Fitting a Graph to Vector Data. *Proceedings: International Conference of Machine Learning*. 2009.

[5] Demaine, Erik. Geometric Folding Algorithms: Linkages, Origami, Polyhedra. Lecture notes and video. Available at: http://courses.csail.mit.edu/6.849/fall10/lectures/. 2010.

[6] Frank, A. & Asuncion, A. (2010). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.

[7] Kanapka, Joseph. Faster Fiedler Vector Computation for Laplacians of Well-Shaped Meshes. Masters Thesis, Massachusetts Institute of Technology. 1998.

[8] Khardon, Roni. Advanced Topics in Machine Learning. Lecture 18. Tufts University. 2008.

[9] Ng, Andrew. CS229 Lecture Notes: Support Vector Machines. http://see.stanford.edu/materials/aimlcs229/cs229-notes3.pdf

[10] Ng, A. Y., Jordan, M. I., & Weiss, Y. On spectral clustering: Analysis and an algorithm. Proc. 15th NIPS (pp. 849856). 2001.

[11] Saul, L. & Roweis, S. An Introduction to Locally Linear Embedding. 2000.

[12] Shalizi, Cosma. Statistics 26-350: Data Mining Lecture Notes. Carnegie Mellon University. 2009.

[13] von Luxburg, Ulrike. A Tutorial on Spectral Clustering. Statistics and Computing. 17(4), 2007.

[14] Zhu, X., Ghahramani, Z., & Lafferty, J. D. Semi-supervised learning using gaussian elds and harmonic functions. Proc. 20th ICML. 2003.

[15] Zhu, Xiaojin. Semi-Supervised Learning with Graphs. Carnegie Mellon University. Thesis. 2005.

Truss Structures and the Sparsity of Graphs Constructed Through Laplacian Function Optimization

Cameron Musco

Advisor: Dan Spielman

December 16, 2011

Abstract

In this report, I describe some of the work I did exploring the sparsity and uniqueness of the hard and soft graphs developed in [2]. I explain the connections found between the structure of the optimization problems used to build these graphs and some results in rigidity theory. Some of this work is also included in the final section of my senior project paper [4].

1 Background and Notation

In [2], the authors propose a graph construction technique in which a graph is built over a set of vectors that minimizes the total error, weighted by node degree, arising from estimating each point as a weighted average of its neighbors. Specifically, they seek edge weights, $\boldsymbol{w}$, that minimize:

$$f(\boldsymbol{w}) = \sum_i \|d_i \boldsymbol{x}_i - \sum_j w_{i,j} \boldsymbol{x}_j\|^2$$

$$f(\boldsymbol{w}) = \|LX\|_F^2$$

where L is the graph laplacian for the graph with edge weights $\boldsymbol{w}$, and $\|M\|_F = (\sum_{i,j} M_{i,j}^2)^{1/2}$ is the Frobenius norm.

Since setting each weighted degree d_i to 0 would clearly minimize this function, the authors introduce two possible methods for restricting the weighted degrees. In the *hard graph* each weighted degree is required to be at least 1. In the *α-soft graph*, the degrees must satisfy the constraint:

$$\sum_i (max(0, 1 - d_i))^2 \leq \alpha n$$

where α is a chosen parameter of the soft graph.

As is described in [2], the above objective function can be reworked to to the form $f(\boldsymbol{w}) = \|M\boldsymbol{w}\|^2$. Let n be the number of vector points, d be the dimension of these vectors, and $m = n(n-1)/2$ be the number of possible edges in a graph over these points. M is an $nd \times m$ matrix with each column corresponding to a possible edge in the graph. It is constructed from d $n \times m$ matrices $M_1, ...M_d$ stacked on top of each other. The column of M_i corresponding to the edge $e = (j, k)$ has all zero entries except for the value $x_j(i) - x_k(i)$ at position j and the value $x_k(i) - x_j(i)$ at position k. We write M out fully just to give a sense of its construction (Latex for matrix taken from [5].):

$$
n \times d \left\{ \overbrace{\left[\begin{array}{cccccccccc}
(x_1(1)-x_2(1)) & (x_1(1)-x_3(1)) & \cdots & (x_1(1)-x_n(1)) & 0 & \cdots & 0 & 0 & \cdots & 0 \\
(x_2(1)-x_1(1)) & 0 & \cdots & 0 & (x_2(1)-x_3(1)) & \cdots & (x_2(1)-x_n(1)) & 0 & \cdots & 0 \\
\vdots & \vdots & & \vdots & \vdots & & \vdots & \vdots & & \vdots \\
0 & 0 & \cdots & (x_n(1)-x_1(1)) & 0 & \cdots & (x_n(1)-x_2(1)) & 0 & \cdots & (x_n(1)-x_{n-1}(1)) \\
\hline
(x_1(2)-x_2(2)) & (x_1(2)-x_3(2)) & \cdots & (x_1(2)-x_n(2)) & 0 & \cdots & 0 & 0 & \cdots & 0 \\
(x_2(2)-x_1(2)) & 0 & \cdots & 0 & (x_2(2)-x_3(2)) & \cdots & (x_2(2)-x_n(2)) & 0 & \cdots & 0 \\
\vdots & \vdots & & \vdots & \vdots & & \vdots & \vdots & & \vdots \\
0 & 0 & \cdots & (x_n(2)-x_2(2)) & 0 & \cdots & (x_n(2)-x_2(2)) & 0 & \cdots & (x_n(2)-x_{n-1}(2)) \\
\hline
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\hline
(x_1(d)-x_2(d)) & (x_1(d)-x_3(d)) & \cdots & (x_1(d)-x_n(d)) & 0 & \cdots & 0 & 0 & \cdots & 0 \\
(x_2(d)-x_1(d)) & 0 & \cdots & 0 & (x_2(d)-x_3(d)) & \cdots & (x_2(d)-x_n(d)) & 0 & \cdots & 0 \\
\vdots & \vdots & & \vdots & \vdots & & \vdots & \vdots & & \vdots \\
0 & 0 & \cdots & (x_n(d)-x_2(d)) & 0 & \cdots & (x_n(d)-x_2(d)) & 0 & \cdots & (x_n(d)-x_{n-1}(d))
\end{array}\right]}^{m} \right.
$$

(The first n rows are marked n.)

2 A lower degree bound

In [2], the authors give an upper bound of $n(d+1)$ on the number of edges in the sparsest solutions to the hard and soft graphs. As explained above, the objective function $f(\boldsymbol{w}) = \sum_i \|d_i \boldsymbol{x}_i - \sum_j w_{i,j}\boldsymbol{x}_j\|^2$ can be written in the form $f(\boldsymbol{w}) = \|M\boldsymbol{w}\|^2$. Let U be the $n \times m$ matrix where each column corresponds to an edge $e(i,j) \in E$ and has all zero entries except $U_{i,e} = 1$ and $U_{j,e} = -1$. Let A be the matrix obtained by taking the absolute values of each element of U. (These matrices are defined in exactly the same way in [2].)

Now, we can see that if there is some $\boldsymbol{e}$ such that $\begin{bmatrix} M \\ A \end{bmatrix} \boldsymbol{e} = 0$ then $M(\boldsymbol{w}+\boldsymbol{e}) = M\boldsymbol{w}$ and $A(\boldsymbol{w}+\boldsymbol{e}) = A\boldsymbol{w}$. $A\boldsymbol{w}$ gives the n length vector of vertex degrees given the edge weights $\boldsymbol{w}$. As explained above, these degrees are restricted to be greater than 1 for the hard graph and to have $\sum_i (max(0, 1-d_i))^2 \leq \alpha n$ for the soft graph. In either case, as long as $\boldsymbol{w}+\boldsymbol{e}$ has no negative entries (an so does not violate the positve edge weight constraint), since it does not change our $M\boldsymbol{w}$ value or vertex degrees, it is an additional valid solution to the optimization problem.

Now, as long as $\boldsymbol{e}$ is 0 wherever $\boldsymbol{w}$ is 0, then we can scale down $\boldsymbol{e}$ so that adding $\boldsymbol{e}$ to $\boldsymbol{w}$ does not lead to any violation of positive edge weights. Further, if we set $c = \min_i |\boldsymbol{w}(i)/\boldsymbol{e}(i)|$ then either $\boldsymbol{w}+c\boldsymbol{e}$ or $\boldsymbol{w}-c\boldsymbol{e}$ will have no negative elements but will have at least one less positive element than $\boldsymbol{w}$. So, specifically, the existence of e tells us that there is some solution $\boldsymbol{w'} = \boldsymbol{w}+c\boldsymbol{e}$, that is sparser than the original solution $\boldsymbol{w}$.

As observed in [2], the rank of $\begin{bmatrix} M \\ A \end{bmatrix}$ is at most $n(d+1)$ - the number of rows in the matrix. So, if we have a solution $\boldsymbol{w}$ with $\geq n(d+1)$ edges (positive entries), then there must be some $\boldsymbol{e}$ with only nonzero entries where $\boldsymbol{w}$ is positive such that $\begin{bmatrix} M \\ A \end{bmatrix} \boldsymbol{e} = 0$. So there must be a sparser solution. So our sparsest solution must have $< n(d+1)$ edges, and hence average degree $< 2(d+1)$.

We can get a slightly better bound by looking more closely at the structure of M. We can split M up into the d matrices $M_1, ...M_d$ shown above. $M_i = UD_i$, where D_i is the $e \times e$ diagonal matrix with $D_i(a,a) = x_j(i) - x_k(i)$ where $e_a = (j,k)$. Now, we can determine the rank of U using a result from [1]. Each column of U, $\boldsymbol{u_k}$, corresponds to an edge $e_k = (i,j)$ of our graph. And $\boldsymbol{u_k}(i) = 1 = -\boldsymbol{u_k}(j)$. All other elements of $\boldsymbol{u_k}$ are 0. Now consider any cycle in our graph made up of edges $e_1e_2...e_i$ passing through the vertices $x_1x_2...x_ix_1$. Then looking at the columns of U corresponding to our edges, $\boldsymbol{u_{e_1}}$ (or $-\boldsymbol{u_{e_1}}$) has a weight of 1 in the x_1 position and a weight of -1 in the x_2 position. $\boldsymbol{u_{e_2}}$ (or $-\boldsymbol{u_{e_2}}$) has a weight of 1 in the x_2 position and a weight of -1 in

the x_3 position. So, we can add $\boldsymbol{u_{e_1}}$ and $\boldsymbol{u_{e_2}}$, with the appropriate positive or negative signs, to get a vector with a 1 at position x_1, a $(-1+1)=0$ at position x_2, a -1 at position x_3, and zeros everywhere else. Call this new vector $\hat{\boldsymbol{u}}$. We can add $\hat{\boldsymbol{u}}$ to $\boldsymbol{u_{e_3}}$ or $-\boldsymbol{u_{e_3}}$ to get a vector with a 1 still at postition x_1 but a 0 at position x_3 and a new -1 at postion x_4. Following this pattern around our cycle, some summation $\hat{\boldsymbol{u}} = c_1 * \boldsymbol{u_{e_1}} + ... + c_{i-1} * \boldsymbol{u_{e_{i-1}}}$ where each c is ± 1 will give us a vector with a 1 at position x_1 and a -1 at postion x_i. So, this summation will give us either $\boldsymbol{u_{e_i}}$ or $-\boldsymbol{u_{e_i}}$ since e_i connects x_1 and x_i, completing our cycle. So, the column of U corresponding to e_i is spanned by the other columns of U corresponding to edges in the cycle $e_1 e_2 ... e_i$. In general, any set of columns chosen such that there is a cycle on the graph using the edges corresponding to those columns will be linearly dependent. A set of columns may only be linearly independent if its corresponding edge set contains no cycles. The largest set of edges containing no cycles is a spanning tree over our graph, which contains $n-1$ edges. So, U has rank $n-1$. Correspondingly U has a null space of dimension 1. This null space is given by scalings of $\mathbf{1}$ (simply summing the rows of U each with weight 1 yields $\mathbf{0}$).

Now, if M were simply d U matrices stacked on top of one another, then $rank(M)$ would be $n-1$. However, each M_i is a U matrix with a different weighting applied to the columns. We may expect that, in general position, the null spaces of each M_i are disjoint, so, we have M with null space of dimension at least d and therefore rank at most $nd-d$. So we have $\begin{bmatrix} M \\ A \end{bmatrix}$ with rank at most $nd-d+n = n(d+1)-d$, which is lower than our original bound.

However, we can do even better than this bound. In fact, each time we add an M_i matrix to the stack, we loose an extra dimension from our rank. i.e. the null space of $\begin{bmatrix} M_i \\ M_j \end{bmatrix}$ has 3 dimensions. This causes us to have in total $rank(M) = (n-1)+(n-2)+...+(n-d) = \boldsymbol{n - [d(d+1)/2]}$. A proof of this fact using rigidity theory is given below.

Theorem 2.1. *The sparsest solution to the hard or soft graph optimization over a set of points has* $< n(d+1) - d(d+1)/2$ *edges.*

Proof. We show this lower rank bound by looking at how M can be viewed as a rigidity matrix, a concept explained in [3]. In rigidity theory, we are given a set of points in d-dimensional space, some of which are joined by rigid trusses. The points do not have fixed locations, however the trusses enforce a constant distance between any two points that they join together. In general, the graph of vertices along with the distance constraints is referred to as a *linkage*. An embedding of the points into d-dimensional space that does not violate the distance constraints is called a *configuration* of the linkage. We want to determine if the structure consisting of the points and the trusses joining them is 'rigid' - that, starting from a valid configuration, there is no non-trival way to rearrange the points ('flex' the structure) while still obeying the distance constraints imposed by the trusses. Trivial flexings include simple shifts or rotations of the points that do not change the overall shape of the structure but do change the locations of the points. We technically define a flexing of the graph as a continuous motion that changes our point locations without violating our distance constraints. An *infinitesimal flexing* is defined to be a set of n instantaneous velocity vectors, which, when applied to our n vertices, do not violate our distance constraints. The existence of a flexing implies the existence of an infinitesimal flexing, as the infinitesimal flexing can be obtained by differentiating the motion that leads to the full flexing.

Now, considering two points in our linkage, which are located at $\boldsymbol{x}$ and $\boldsymbol{y}$ in a valid configuration and are joined by a truss with a given length l, in order to not violate the distance constraint imposed by the truss, if $\boldsymbol{\delta_x}$ is our instantaneous velocity vector on $\boldsymbol{x}$ and $\boldsymbol{\delta_y}$ our velocity vector for $\boldsymbol{y}$, then we must have:

$$\begin{aligned} l = \|\boldsymbol{x}-\boldsymbol{y}\| &= \|[\boldsymbol{x}+\boldsymbol{\delta_x}]-[\boldsymbol{y}+\boldsymbol{\delta_y}]\| \\ \|\boldsymbol{x}-\boldsymbol{y}\| &= \|[\boldsymbol{x}-\boldsymbol{y}]-[\boldsymbol{\delta_y}-\boldsymbol{\delta_x}]\| \\ (\boldsymbol{x}-\boldsymbol{y})\cdot(\boldsymbol{x}-\boldsymbol{y}) &= (\boldsymbol{x}-\boldsymbol{y})\cdot(\boldsymbol{x}-\boldsymbol{y}) - 2(\boldsymbol{x}-\boldsymbol{y})\cdot(\boldsymbol{\delta_x}-\boldsymbol{\delta_y}) + (\boldsymbol{\delta_x}-\boldsymbol{\delta_y})\cdot(\boldsymbol{\delta_x}-\boldsymbol{\delta_y}) \end{aligned}$$

And, remembering that $\boldsymbol{\delta_x}$ and $\boldsymbol{\delta_y}$ are instantaneous velocity vectors, $(\boldsymbol{\delta_x} - \boldsymbol{\delta_y}) \cdot (\boldsymbol{\delta_x} - \boldsymbol{\delta_y}) = \|\boldsymbol{\delta_x} - \boldsymbol{\delta_y}\|^2$ goes to 0, so we have:

$$\begin{aligned}(\boldsymbol{x} - \boldsymbol{y}) \cdot (\boldsymbol{x} - \boldsymbol{y}) &= (\boldsymbol{x} - \boldsymbol{y}) \cdot (\boldsymbol{x} - \boldsymbol{y}) - 2(\boldsymbol{x} - \boldsymbol{y}) \cdot (\boldsymbol{\delta_x} - \boldsymbol{\delta_y}) \\ 0 &= (\boldsymbol{x} - \boldsymbol{y}) \cdot (\boldsymbol{\delta_x} - \boldsymbol{\delta_y}) \\ 0 &= \sum_{i=1}^{d} [\boldsymbol{x}(i) - \boldsymbol{y}(i)] * [\boldsymbol{\delta_x}(i) - \boldsymbol{\delta_y}(i)]\end{aligned}$$

Now, letting m be the number of possible trusses (edges) between our n points, we can define the rigidity matrix R_i to be the $m \times n$ matrix to be the matrix where each row corresponds to an edge $e_{j,k}$ and has all zero entries except contains the value $\boldsymbol{x}(j) - \boldsymbol{y}(k)$ in the j^{th} position and $\boldsymbol{x}(k) - \boldsymbol{y}(k)$ in the k^{th} position. Letting $\boldsymbol{\Delta_i} = [\boldsymbol{\delta_1}(i)\boldsymbol{\delta_2}(i)...\boldsymbol{\delta_n}(i)]$, (i.e. the vector with the i^{th} dimension component of each of our velocity vectors) we can see that, if e is the edge connecting vertices j and k,

$$\begin{aligned}R_i\boldsymbol{\Delta_i}(e) &= (\boldsymbol{x_j}(i) - \boldsymbol{x_k}(i))\boldsymbol{\delta_j}(i) + (\boldsymbol{x_k}(i) - \boldsymbol{x_j}(i))\boldsymbol{\delta_k}(i) \\ R_i\boldsymbol{\Delta_i}(e) &= [\boldsymbol{x_j}(i) - \boldsymbol{x_k}(i)] * [\boldsymbol{\delta_j}(i) - \boldsymbol{\delta_k}(i)]\end{aligned}$$

And, letting our full rigidity matrix be $R = [R_1...R_d]$ and $\boldsymbol{\Delta} = [\boldsymbol{\Delta_1}...\boldsymbol{\Delta_d}]$ we have:

$$R\boldsymbol{\Delta}(e) = \sum_{i=1}^{d} ([\boldsymbol{x_j}(i) - \boldsymbol{x_k}(i)] * [\boldsymbol{\delta_j}(i) - \boldsymbol{\delta_k}(i)])$$

So, $\delta_1, ...\delta_n$ are an infinitesimal flexing iff $R\boldsymbol{\Delta} = 0$, so $\boldsymbol{\Delta}$ is in the null space of R. Now, R is exactly the same as M^T as defined above. And, the null space of R is well studied in rigidity theory. Specifically, at a minimum, even if a structure is rigid, we can change the configuration of the points with a translation, rotation, or combination of the two. There are d possible translations in d-dimensional space. There are $\binom{d}{2} = \frac{d(d-1)}{2}$ possible rotations. So, our null space, which includes any combinations of these rotations and translations has dimension $d + \frac{d(d-1)}{2} = \frac{d(d+1)}{2}$.

Correspondingly, $R = M^T$ has maximum rank $n * d - d(d+1)/2$. And, since $rank(M) = rank(M^T)$, this means that M has the same maximum rank. And, so $\begin{bmatrix} M \\ A \end{bmatrix}$ has maximum rank $n(d+1) - d(d+1)/2$.

So, by the same argument given in [2], the sparsest solution to the hard or soft graph optimizations over a set of points has $< n(d+1) - d(d+1)/2$ edges. Of course, this bound is not significantly lower than the $n(d+1)$ bound, unless d is large compared to n.

□

3 Other relationships with truss structures

Another interesting application of rigidity theory to the study of the hard and soft graphs is in looking at conditions that hold for optimal graphs. Let an *unconstrained edge* be any edge in a hard graph that is connected to 2 vertices with degree > 1. In order for a hard graph to be an optimal solution to our objective function, we must have that $\frac{df}{d\boldsymbol{w}}$ is 0 on all unconstrained edges included in the graph - so increasing or decreasing edge weights cannot improve the objective function, and is ≥ 0 on all edges not included in the graph - so increasing the weights on these currently zero-weighted edges cannot decrease the objective function.

Recall that our objective function is $f(\boldsymbol{w}) = \sum_i \|d_i\boldsymbol{x}_i - \sum_j w_{i,j}\boldsymbol{x}_j\|^2 = \sum i\|\boldsymbol{\delta_i}\|^2$ where we define $\boldsymbol{\delta_i} = d_i\boldsymbol{x}_i - \sum_j w_{i,j}\boldsymbol{x}_j$. It is the vector giving the degree weighted difference between the true value $\boldsymbol{x_i}$ and the value reconstructed as a weighted average of $\boldsymbol{x_i}$'s neighbors.

Now, consider an unconstrained edge included in a hard graph: $e = (i, j)$. In order for an infinitesimally small change δ in the weight of e to not decrease the value of the objective function, we must have:

$$\begin{aligned} \|\boldsymbol{\delta_i}\|^2 + \|\boldsymbol{\delta_j}\|^2 &\leq \|\boldsymbol{\delta_i} + \delta * \boldsymbol{x_i} - \delta * \boldsymbol{x_j}\|^2 + \|\boldsymbol{\delta_j} + \delta * \boldsymbol{x_j} - \delta * \boldsymbol{x_i}\|^2 \\ \boldsymbol{\delta_i} \cdot \boldsymbol{\delta_i} + \boldsymbol{\delta_j} \cdot \boldsymbol{\delta_j} &\leq \boldsymbol{\delta_i} \cdot \boldsymbol{\delta_i} + \boldsymbol{\delta_j} \cdot \boldsymbol{\delta_j} + 2\delta(\boldsymbol{\delta_i} - \boldsymbol{\delta_j}) \cdot (\boldsymbol{x_i} - \boldsymbol{x_j}) + \delta^2(\boldsymbol{x_i} - \boldsymbol{x_j}) \cdot (\boldsymbol{x_i} - \boldsymbol{x_j}) \\ 0 &\leq \delta(\boldsymbol{\delta_i} - \boldsymbol{\delta_j}) \cdot (\boldsymbol{x_i} - \boldsymbol{x_j}) \end{aligned}$$

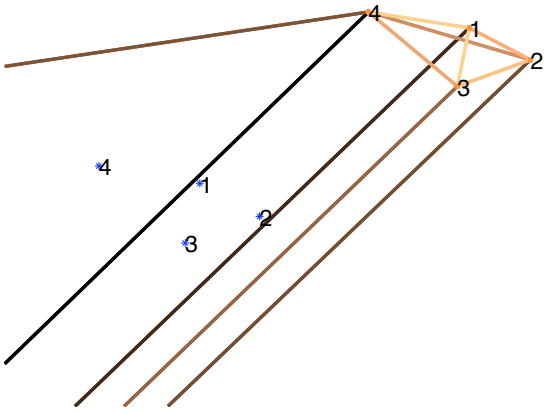
since the δ^2 term goes to 0. Further, since we cannot have a change of $-\delta$ changing the objective function either, we have:

$$\begin{aligned} 0 &= \delta(\boldsymbol{\delta_i} - \boldsymbol{\delta_j}) \cdot (\boldsymbol{x_i} - \boldsymbol{x_j}) \\ 0 &= (\boldsymbol{\delta_i} - \boldsymbol{\delta_j}) \cdot (\boldsymbol{x_i} - \boldsymbol{x_j}) \end{aligned}$$

This is exactly the requirement obtained above for instantaneous velocities on the i^{th} and j^{th} nodes that still preserve edge lengths in a truss structure. For edges not included in the graph, we do not need to worry about the effect of a change in $-\delta$ since we can only increase the weight on non-included edges. So for these edges, we only need $0 \leq (\boldsymbol{\delta_i} - \boldsymbol{\delta_j}) \cdot (\boldsymbol{x_i} - \boldsymbol{x_j})$. This condition exactly corresponds to the condition for instantaneous velocities on the i^{th} and j^{th} nodes not decreasing edge length. So, this condition is the same as the rigidity condition if the nodes are connected by a strut rather than a truss - where the strut can increase, but not decrease, in length. (See [3] for more discussion of rigidity theory as applied to struts and cables along with trusses.)

So, for a set of unconstrained vertices in a hard graph, the vector of degree weighted errors between the true vectors and the neighbor estimated vectors must be a scaling of an instantaneous velocity vector for a valid flexing of the points, where the points connected in the graph are connected by trusses and the points not connected in the graph are connected by struts. If this linkage of the points is rigid, then the weighted error vector must be a scaling of an instantaneous velocity vector for a translation, rotation, or combination of the two. The figure below demonstrates this fact by showing that the degree weighted error vectors for four unconstrained vertices in a hard graph are in fact scalings of the velocity vector for a translation of those vertices.

Figure 1: Blue stars represent the locations of weighted error vectors beginning at the true vertices. The error vectors clearly point in the direction of a shift of the unconstrained vertices.



Of course, the above analysis only applies to hard graphs and only applies to sets of unconstrained vertices. Sets of even more than two connected unconstrained vertices are extremely rare in natural data, as the majority of vertices in the hard graph tend to abut the degree constraint. It is worth

noting that, the counter example for uniqueness and planarity of the hard and soft graphs that Christopher Musco and I found contains a set of four unconstrained vertices connected in clique.

References

[1] Chen, Doron. Algebraic and Algorithmic Applications of Support Theory. Tel-Aviv University. 2005.

[2] Daitch, S., Kelner, J., & Spielman, D. Fitting a Graph to Vector Data. *Proceedings: International Conference of Machine Learning*. 2009.

[3] Demaine, Erik. Geometric Folding Algorithms: Linkages, Origami, Polyhedra. Lecture notes and video. Available at: http://courses.csail.mit.edu/6.849/fall10/lectures/. 2010.

[4] Musco, Cameron. Graph Construction Through Laplacian Function Optimization. Senior Project Report. Yale University. 2011.

[5] Musco, Christopher. Graph Construction for Machine Learning - Optimization and Combinatorial Properties. Senior Project Report. Yale University. 2011.

Counterexample to Uniqueness and Planarity of Constructed Graphs

Christopher Musco and Cameron Musco

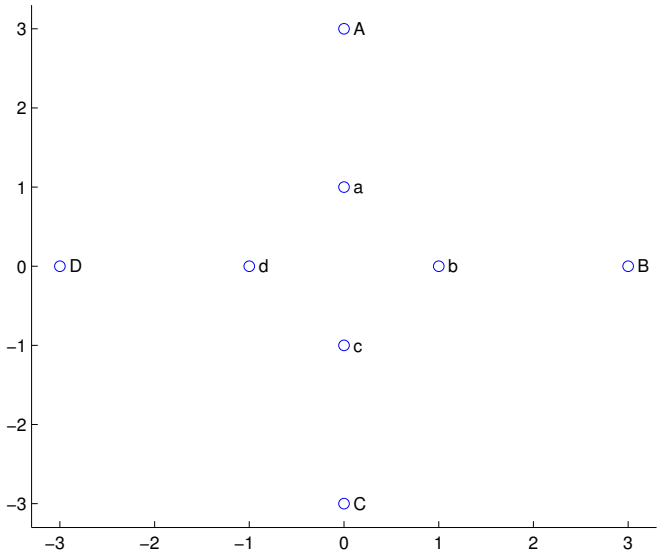
December 29, 2011

1 Conjecture

It was conjectured in the 2009 paper, "Fitting a Graph to Vector Data," by Samuel Daitch, Jonathan Kelner, and Daniel Spielman [DKS09] that the hard and soft objective functions introduced, which are minimized to construct graphs for learning, lead to unique graphs on points in general position. Specifically, the authors suggest that, under arbitrarily small perturbations, any set of points will admit a unique solution with probability 1. This conjecture seemed very reasonable: randomly generated and real data had always led to unique solutions and any point sets found with multiple minimizing graphs were highly symmetric. Under small perturbations, the non-uniqueness always collapsed with the elimination of symmetry (see [ChM11] for a simple example). As such, we spent quite a bit of the Fall 2011 semester trying to prove that the objective functions do in fact lead to uniqueness. However, our efforts ultimately ended in the discovery of several counter examples that did not admit unique solutions, even under perturbation. The simplest set of points found is presented below.

2 Counter Example

Figure 1: Counter Exampe in 2 Dimensions



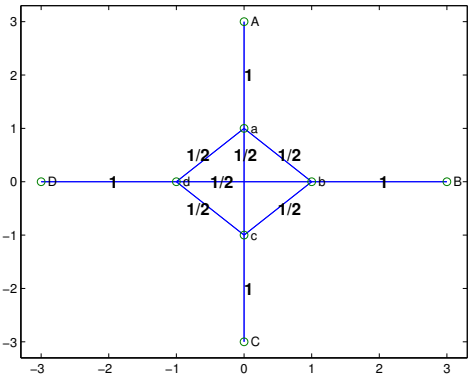
We'll use the labels in the above diagram in our discussion below. Just to be clear, our set of 8 vectors consists of: $A = [0, 3]$, $a = [0, 1]$, $B = [3, 0]$, $b = [1, 0]$, $C = [0, -3]$, $c = [0, -3]$, $D = [-3, 0]$, and $d = [-1, 0]$.

2.1 Hard Graph Solutions Under Symmetry

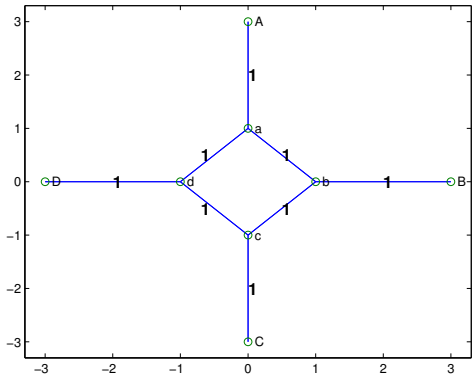
For now, lets restrict our discussion to the slightly simpler hard constraints, which require that each vertex has degree ≥ 1. The point set above has several optimal hard graphs, including the following two:

Figure 2: Symmetric Solutions

(a) Non-planar



(b) Planer



Recall that the function we are seeking to minimize is:

$$f(G) = \sum_{i=1}^{n} \left|\left| deg_i \boldsymbol{x}_i - \sum_{j=1}^{n} w_{i,j} \boldsymbol{x}_j \right|\right|^2 = \sum_{i=1}^{n} \left|\left| deg_i \left(\boldsymbol{x}_i - \sum_{j \neq i} \frac{w_{i,j}}{deg_i} \boldsymbol{x}_j \right) \right|\right|^2$$

To see that each of these solutions is optimal, consider the inner set of points, $a, b, c,$ and d and the outer set, $A, B, C,$ and D. In both solutions, each inner point is perfectly predicted by the weighted average of its neighbors. For example, in the non-planar solution, (a):

$$\frac{1 * A + \frac{1}{2} * b + \frac{1}{2} * c + \frac{1}{2} * d}{deg_a} = \frac{[0, 3] + \frac{1}{2}[1, 0] + \frac{1}{2}[0, -3] + \frac{1}{2}[0, -3]}{2\frac{1}{2}} = [0, 1] = a$$

And in the planar solution, (b):

$$\frac{1 * A + 1 * b + 1 * d}{deg_a} = \frac{[0, 3] + [1, 0] + [0, -3]}{3} = [0, 1] = a$$

Thus, regardless of its weighted degree, each inner point contributes no error to $f(G)$. The outer points on the other hand do contribute error. However, the prediction error for each is 2 (the distance to the corresponding inner point), which is the minimum distance between an outer point and any convex combination (weighted average) of the other 7 points. Combined with the fact that each outer point has minimum degree, this ensures that each contributes as little as possible to the objective function. Thus, both graphs above minimize $f(G)$ at $4 * (1 * 2)^2 + 4 * 0^2 = \mathbf{16}$.

2.2 Connection Between Uniqueness and Planarity

Before discussing the above example further, it is worth noting that, in two dimensions, if it is possible to find a non-planar graph that minimizes our objective function, the graph is not a unique solution. This follows from **Theorem 3.3** in [DKS09], which demonstrates that it is always possible to eliminate intersecting cliques in graphs without changing the value of the objective function over the graph and without decreasing any vertex degrees. Thus, eliminating intersecting cliques in a solution graph leads to a new graph that will not violate the hard or soft degree constraints. A non-planar graph contains at least one pair of intersecting edges (2-cliques) and thus, if we have a non-planar solution, we can find a new solution by eliminating one of the intersecting edges.

In fact, when eliminating edges from intersecting cliques, the degrees of all vertices in the cliques increase strictly. Generally, this prevents us from running the process described in the proof for **Theorem 3.3** in reverse to add intersecting cliques to graphs without affecting our objective function. If we did, we might violate degree constraints that are tight to begin with ($deg_i = 1$). However, this is not the case when we have a group of vertices all with high degree. For example, in two dimensions, if a set of edges form a quadrilateral, the vertices of which all have degree > 1, it is always possible to increase the weight on both diagonals of the quadrilateral (which may have 0 weight to begin with) without changing the value of $f(G)$. The edge weights on the perimeter of the quadrilateral will decrease, but as long as we choose a low enough value for r (using the notation from [DKS09]), this will not cause a problem as long as non-zero to begin with. Thus, in two dimensions, finding a cycle of 4 points with degrees > 1 is sufficient for demonstrating that a solution is not unique.

2.3 Non-uniqueness for Under Perturbation

We continue to look only at the hard graph. Consider the set of points generated as a small perturbation of the points given above. That is, renaming the outer points A_{old}, B_{old}, C_{old}, D_{old} and the inner points a_{old}, b_{old}, c_{old}, and d_{old}, call our new points A, B, C, D, a, b, c, and d, where each point is located at any position in a circle of radius δ around its corresponding original point for some $\delta > 0$. In order to simplify the following proof that, for small enough δ, the hard graph over the perturbed points is non-unique, we assume that the outer points are only pushed in an outward direction when perturbed and that the inner points are only pushed inwards. That is, letting $\boldsymbol{\delta_v}$ be the shift from v_{old} to v, $\boldsymbol{\delta_A}$ has a positive y component while $\boldsymbol{\delta_a}$ has a negative y component. $\boldsymbol{\delta_B}$ has a positive x component while $\boldsymbol{\delta_b}$ has a negative x component, etc.

Even with this restriction on the perturbations, the perturbed points are still in general position. As long as we can show non-uniqueness under these restrictions, we will have successfully demonstrated that the above set of points does not admit a unique solution with probability 1 when perturbed.

Let $\boldsymbol{w'}$ be the weight vector corresponding to the planar solution, (b), given above. Intuitively, given a small shift of our original points, a solution similar to $\boldsymbol{w'}$ will still be a solution to the optimization problem. In an optimal solution, to have a reasonably small total error from approximating the points as weighted sums of their neighbors, we must still have weights of near 1 connecting each outer point to its nearest inner point. Additionally, we cannot have significant weight connecting the outer point to other points in the graph. To balance out the strong radial inner-outer connections, we must still have strong connections between the inner points, leading to solutions that either 1) include both edge ac and edge bd so are non-planar, or 2) that contain the full quadrilateral $abcd$ with vertex degrees > 1. As explained above, in both of these cases, our solution will be non-unique.

If $\delta << 1$, then the y coordinate of A, A_y, is more than 2 greater than v_y for any other point v. Since $A'_y = a'_y + 2$ and A shifts up while a shifts down, $A_y - a_y > 2$. Symmetrically, B_x is more than 2 greater than any v_x, C_y is more than 2 less than any v_y, and D_x is more than 2 less than any v_x. So, each outer point is at least a distance of 2 away from the convex hull of the other 7 points. Thus, since each vertex must have degree at least 1, the error component of $f(\boldsymbol{w})$ corresponding to each

of A, B, C, and D is at least $2^2 = 4$ for any $\boldsymbol{w}$. Even if a, b, c, and d can be perfectly reconstructed as weighted sums of their neighbors, if ϵ is the minimum value of $f(\boldsymbol{w})$ over all possible $\boldsymbol{w}$, we have that $\epsilon \geq 4 * 4 = 16$.

Now, we can also put an upper bound on ϵ by applying the solution (b) to our perturbed graph. In (b), we see that A is approximated only by a, which differs by at most $2 + 2\delta$ in the y direction and 2δ in the x direction. So, f_A, the component of our error corresponding to A, is at most $\|[2\delta, 2 + 2\delta]\|^2 = (2\delta)^2 + (2 + 2\delta)^2$. The y error for a is maximized when a is shifted down while A, b, and d are shifted up. So, instead of being exactly approximated as above, it is off by at most $\frac{2\delta * 3}{3} = 2\delta$. The x error for a is also maximized when a is shifted in the opposite x direction of its neighbors. Thus, the x error is at most 2δ. So, $f_a \leq (2\delta)^2 + (2\delta)^2$. Since the above arguments apply symmetrically to the other inner and outer points, we have

$$\begin{aligned}\epsilon &\leq 4 * [(2 + 2\delta)^2 + (2\delta)^2] + 4 * [(2\delta)^2 + (2\delta)^2] \\ &\leq 16 + (32\delta + 64\delta^2)\end{aligned}$$

Now, consider some $\boldsymbol{w}$ that minimizes f over our perturbed graph (so $f(\boldsymbol{w}) = \epsilon$). Looking at A, f_A is at least the weighted squared error in the y direction. Since A_y is at least 2 greater than a_y, and at least $3 - 2\delta$ greater than v_y for all other v, we have:

$$f_A \geq (2w_{A,a} + (3 - 2\delta) * \sum_{v \neq a} w_{A,v})^2$$

where $w_{i,j}$ is the weight on the edge between vertex i and vertex j. However, as described when giving the lower bound above, each of B, C, and D contribute at least 4 to f, so $f_A(\boldsymbol{w}) \leq f(\boldsymbol{w}) - 4 * 3 \leq 4 + (32\delta + 64\delta^2)$. So:

$$2w_{A,a} + (3 - 2\delta) * \sum_{v \neq a} w_{A,v} \leq \sqrt{4 + (32\delta + 64\delta^2)}$$

$$2w_{A,a} + (3 - 2\delta)(1 - w_{A,a}) \leq \sqrt{4 + (32\delta + 64\delta^2)}$$

since $deg_A \geq 1$, and thus $\sum_{v \neq a} w_{A,v} \geq 1 - w_{A,a}$. So we have:

$$\begin{aligned}2w_{A,a} + 3 - 2\delta - 3w_{A,a} + 2\delta w_{A,a} &\leq 2 + 8\delta \\ 1 - 10\delta &\leq w_{A,a}(1 - 2\delta) \\ \frac{1 - 10\delta}{1 - 2\delta} &\leq w_{A,a}\end{aligned}$$

Similarly, letting S equal $\sum_{v \neq a} w_{A,v}$, we have that $w_{A,a} \geq 1 - S$ so:

$$\begin{aligned}2(1 - S) + (3 - 2\delta)S &\leq \sqrt{4 + (32\delta + 64\delta^2)} \\ 2 + (1 - 2\delta)S &\leq 2 + 8\delta \\ S &\leq \frac{8\delta}{1 - 2\delta}\end{aligned}$$

Finally, just to give an upper bound on deg_A, we know that we must have:

$$\begin{aligned}deg_A * 2 &\leq \sqrt{4 + (32\delta + 64\delta^2)} \\ deg_A &\leq 1 + 4\delta\end{aligned}$$

Thus, if $\delta < .001$, then $S < 0.008$, $w_{A,a} > 0.991$, and $deg_A < 1.004$. The same bounds hold symmetrically for the other outer points, B, C, and D. Essentially this tells us that, as in (b), in order to have an optimal construction, each outer point must be connected very strongly to its nearest inner point.

Now, consider the inner point a. Since A, B, C, and D contribute at least a weighted error of 4 each to $f(\boldsymbol{w})$, we have that the error contributed by a is $f_a(\boldsymbol{w}) \leq \epsilon - 4 * 4 \leq 32\delta + 64\delta^2$. Since the total weighted error due to a is at least equal to the total error in the y direction, we just consider this y error. From above, we have that for $\delta < .001$, $w_{A,a} \geq 0.991$. In order to balance out the effect of this edge, which causes a high approximation for the y value of a, there must be significant weight on edges from a to the other points in the graph (which all have lower y value than a and A). Additionally, since we know from above that S is small and thus a cannot be strongly connected to either B, C, or D, the majority of this weight must fall on the edges between a and its fellow inner points. Specifically, let $S' = \sum_{v \in \{b,c,d\}} w_{a,v}$, and let $\Delta_{v,a} = v_y - a_y$ for any other point v. Then we must have:

$$\left| \sum_v \Delta_{v,a} * w_{v,a} \right| \leq \sqrt{32\delta + 64\delta^2}$$
$$\sum_v \Delta_{v,a} * w_{v,a} \leq \sqrt{32\delta + 64\delta^2}$$

We know that $\Delta_{A,a} \geq 2$, $0 > \Delta_{D,a} \geq -1 - 2\delta$, $0 > \Delta_{B,a} \geq -1 - 2\delta$, and $0 > \Delta_{C,a} \geq -4 - 2\delta$. Again we know from above that, when $\delta < 0.001$, $w_{a,A} \geq 0.991$ while $w_{B,a}$, $w_{C,a}$, and $w_{D,a}$ are all $\leq S < 0.008$. So we have:

$$0.991 * 2 + 2 * 0.008 * (-1 - 2\delta) + 0.008 * (-4 - 2\delta) + \sum_{v \in \{b,c,d\}} (\Delta_{v,a} * w_{v,a}) \leq \sum_v \Delta_{v,a} * w_{v,a}$$
$$\leq \sqrt{32\delta + 64\delta^2}$$
$$(1.93 - 0.05\delta) + \sum_{v \in \{b,c,d\}} (\Delta_{v,a} * w_{v,a}) \leq \sqrt{32\delta + 64\delta^2}$$

For $\delta < 0.001$, we know that $1.93 - 0.05\delta > 1.9$ and $\sqrt{32\delta + 64\delta^2} < 0.18$. Additionally, we know that $\Delta_{v,a} \geq (-1 - 2\delta)$ for $v = b$ or $v = d$ and $\Delta_{c,a} \geq -2 - 2\delta$. Thus:

$$1.9 + w_{a,b} * (-1 - 2\delta) + w_{a,d} * (-1 - 2\delta) + w_{a,c} * (-2 - 2\delta) < 0.18$$
$$w_{a,b} * (1 + 2\delta) + w_{a,d} * (1 + 2\delta) + w_{a,c} * (2 + 2\delta) > 1.7$$
$$w_{a,b} * 1.002 + w_{a,d} * 1.002 + w_{a,c} * 2.002 > 1.7$$

The above equation holds symmetrically for b, c, and d. Consider 2 cases:

1: $w_{a,c} > 0$ and $w_{b,d} > 0$

In this case, $\boldsymbol{w}$ gives a non-planar graph since it places positive weight on the crossing edges $w_{a,c}$ and $w_{b,d}$. Thus our solution is non-planar and non-unique, so we are done.

2: $w_{a,c} = 0$ or $w_{b,d} = 0$

Consider the case when $w_{a,c} = 0$. The same results will apply symmetrically when $w_{b,d} = 0$. We must have $1.7 < w_{a,b} * 1.002 + w_{a,d} * 1.002$ and $1.7 < w_{c,b} * 1.002 + w_{c,d} * 1.002$. Considering just a first, dividing through by 1.002 and rounding down, we see that:

$$1.6 < w_{a,b} + w_{a,d}$$

Now, we could place all of the required 1.6 weight on one of $w_{a,b}$ or $w_{a,d}$, but this would cause the approximation of a in the x direction to be off by too much. As discussed above in regards to y deviation, it must be that the approximation of a in the x direction, weighted deg_a, is off by less than $\sqrt{32\delta + 64\delta^2}$. Adopting $\Delta_{v,a}$ to equal $v_x - a_x$ this means we must have:

$$\sum_v \Delta_{v,a} * w_{v,a} \leq \sqrt{32\delta + 64\delta^2}$$

Recall that $\Delta_{A,a} \geq -2\delta$, $w_{A,a} \leq deg_A \leq 1.004$, $\Delta_{C,a} \geq -2\delta$, $w_{C,a} \leq 0.008$, $\Delta_{D,a} \geq -3-2\delta$, $w_{D,a} \leq 0.008$, $\Delta_{ba} \geq 1-2\delta$, and $\Delta_{da} \geq -1-2\delta$. So, even if we assign $w_{B,a} = 0$ and A_{old}, C_{old}, and D_{old} are all shifted left, with maximum weight placed on edges Aa, Ca, and Da, to help balance the rightwards deviation of b from a, we still have:

$$(1.004 + 0.008) * -2\delta + 0.008 * (-3 - 2\delta) + w_{a,b} * (1 - 2\delta) + w_{a,d} * (-1 - 2\delta) \leq \sqrt{32\delta + 64\delta^2}$$
$$(-0.024 - 2.04\delta) + w_{a,b} * (1 - 2\delta) + w_{a,d} * (-1 - 2\delta) \leq \sqrt{32\delta + 64\delta^2}$$

For $\delta < 0.001$ this gives:

$$-0.03 + w_{a,b} * 0.998 + w_{a,d} * (-1.002) < 0.18$$
$$0.998 * w_{a,b} - 1.002 * w_{a,d} < 0.19$$

If $w_{a,d} \leq 0.5$ then we have $0.998 * w_{a,b} - 1.002 * w_{a,d} \geq 0.998 * (1.6 - .5) - 1.002 * 0.5 > 0.59$. So the above equation does not hold. So, in order to approximate a reasonably well in the x direction we must have $w_{a,d} > 0.5$. We can use a symmetric argument to show that we also must have $w_{a,b} > 0.5$. Further, we can use symmetric arguments focusing on c instead of a to show that $w_{c,b} > 0.5$ and $w_{c,d} > 0.5$. So, we have that, if $w_{a,c} = 0$, then the weight on each edge of the quadrilateral *abcd* in our graph is greater than 0.5. This means that each of a, b, c, and d have degree > 1, so as described earlier in the paper, it must be that our optimal solution is not unique. We can (again symmetrically) show that the same fact holds if $w_{b,d} = 0$. So, **if we have an optimal solution in which either $w_{a,c} = 0$ or $w_{b,d} = 0$, then this solution is not unique**.

With the above two cases we have shown that, given a set of points in general position generated as a random perturbation of the points in our original counter example, the solution to the hard graph optimization problem is non-unique with some positive probability. □

2.4 Extension of Results to Soft Graph

We do not explicitly extend the above results to the soft-graph optimization problem. However, if α is low enough, it is clear that the optimal soft graph will not deviate significantly from the optimal hard graph. So, there will be an optimal soft graph with either positive weight on both *ac* and *bd* or the full path *abcd* with vertex degrees > 1, which leads to non-uniqueness. It would be nice to show that a counter example exists for all α, which we suspect is true if the inner quadrilateral in the example given above is scaled down as α increases.

References

[DKS09] Daitch, S., Kelner, J., & Spielman, D. Fitting a Graph to Vector Data. *Proceedings: International Conference of Machine Learning*. 2009.

[CaM11] Musco, Cameron. Graph Construction Through Laplacian Function Optimization. *Yale College: Senior Project Report*. 2011.

[ChM11] Musco, Christopher. Graph Constructions for Machine Learning: Optimization and Combinatorial Properties. *Yale College: Senior Project Report*. 2011.