Linear-Space Algorithms for Distance Preserving Embedding*

Tetsuo Asano[1] Prosenjit Bose[2] Paz Carmi[2] Anil Maheshwari[2] Chang Shu[3] Michiel Smid[2]
Stefanie Wuhrer[2,3]

Abstract

The *distance preserving graph embedding problem* is to embed vertices of a given weighted graph into points in 2-dimensional Euclidean space so that for each edge the distance between their corresponding endpoints is as close to the weight of the edge as possible. If the given graph is complete, that is, if distance constraints are given as a full matrix, then principal coordinate analysis can solve it in polynomial time. A serious disadvantage is its quadratic space requirement. In this paper we develop linear-space algorithms for this problem. A key idea is to partition a set of n objects into disjoint subsets (clusters) of size $O(\sqrt{n})$ such that the minimum inter cluster distance is maximized among all possible such partitions.

1 Introduction

Suppose a set S of n objects is given and for each pair of objects (i, j) their dissimilarity, denoted by $\delta_{i,j}$, can be computed in constant time. Using the dissimilarity information, we want to map objects into points in a low dimensional space so that the dissimilarities are preserved as the distances between the corresponding points.

Converting distance information into coordinate information is helpful for human perception because we can see how close two objects are. Problems where we wish to embed dissimilarities as points in a low dimensional space arise in many different settings including stock market problems [5], computer graphics [6] and computer vision [2].

Multi-Dimensional Scaling (MDS) [3] is a general and powerful framework for constructing a configuration of points in a low dimensional space using information on $\delta_{i,j}$. Given a set of n objects in high-dimensional space as well as the pairwise dissimilarities $\delta_{i,j}, 1 \leq i, j \leq n$ with $\delta_{i,j} = \delta_{j,i}$, the aim is to find a set $P(S)$ of points $p_1, \ldots, p_n$ in d-dimensional space, such that the Euclidean distance between p_i and p_j equals $\delta_{i,j}$. Since this aim can be shown to be too ambitious, we aim to find a good approximation. Different related optimality measures can be used to reach this goal.

Classical MDS, also called Principal Coordinate Analysis (PCO), assumes that the dissimilarities are Euclidean distances in a high-dimensional space and aims to minimize

$$E_{PCO} = \sum_{i,j} |d(p_i, p_j)^2 - \delta_{i,j}^2| \tag{1}$$

Equation 1 is optimized by computing the d largest eigenvalues and corresponding eigenvectors of the distance matrix [3]. Thus, if we neglect some numerical computational issues concerning eigenvalue computation, the PCO embedding for n objects can be computed in polynomial time in n using quadratic space. Note that PCO's result can be poor when the $(d+1)$-st largest eigenvalue is not negligible compared to the d-th largest one for embedding into d-space.

Least-squares MDS (LSMDS) aims to minimize

$$E_{LSMDS} = \sum_{i,j} (\delta_{i,j} - d(p_i, p_j))^2 \tag{2}$$

Equation 2 can be solved numerically by using scaling by maximizing a convex function [3]. However, the algorithm can get stuck in local minima E_{LSMDS}. The embedding can be computed in polynomial time in n using quadratic space.

Although PCO and LSMDS are powerful techniques, they have serious drawbacks for practical use, that is, their high space complexity, since the input is an $n \times n$ matrix specifying pairwise dissimilarities (or distances). In this paper, we present a method for dimensionality reduction that avoids this high space complexity if the dissimilarity information is given by a function that can be evaluated in constant time. A key idea is to use clustering. We propose a simple algorithm for finding a size-constrained clustering and show that our solution achieves largest inter-cluster distance, or maximizes the smallest distance between objects from different clusters. That is, given a set of n objects, with a function evaluating dissimilarities for pairs of objects, we partition the set into $O(\sqrt{n})$ disjoint subsets, called *clusters*, where each cluster contains $O(\sqrt{n})$ objects. Since, each

*Research supported in part by the Ministry of Education, Science, Sports and Culture, Grant-in-Aid for Scientific Research on Priority Areas, Scientific Research (B), NSERC, NRC, MITACS, and MRI. We thank attendees of the 9th Korean Workshop on CG and Geometric Networks 2006. Work by T.A. was done in 2006 while visiting MPI, Carleton University, and NYU.

[1]Japan Advanced Institute of Science and Technology
[2]Carleton University, Ottawa, Canada
[3]National Research Council of Canada

cluster has a relatively small number of objects, and thus performing MDS with a distance matrix for each cluster separately requires only $O(n)$ working space. Using this we devise linear space algorithms for embedding all the objects in the plane.

2 Clustering

Let S be a set of n objects: $S = \{1, \ldots, n\}$. We assume we are given a function which computes the *dissimilarity* between any pair (i, j) of objects as δ_{ij} with $\delta_{ii} = 0$ and $\delta_{ij} = \delta_{ji}$. A partition $\mathcal{P}$ of a set S into k disjoint clusters $C_1, \ldots, C_k$ is called a *k-partition of S*. A k-partition $\mathcal{P}$ is characterized by two distances, *inner-cluster distance* $D_{inn}(\mathcal{P})$ and *inter-cluster distance* $D_{int}(\mathcal{P})$, which are defined by $D_{inn}(\mathcal{P}) = \max_{C_i \in \mathcal{P}} \max_{p,q \in C_i} d_{pq}$, and $D_{int}(\mathcal{P}) = \min_{C_i \neq C_j \in \mathcal{P}} \min_{p \in C_i, q \in C_j} d_{pq}$. When we define a complete graph $G(S)$ with edge weights being dissimilarities, edges are classified into inner-cluster edges interconnecting vertices of the same cluster and inter-cluster edges between different clusters. The inner-cluster distance is the largest weight of inner-cluster edges and the inter-cluster distance is the smallest weight of inter-cluster edges.

A k-partition is called *farthest* (*most compact*, respectively) if it is a k-partition with largest inter-cluster distance (smallest inner-cluster distance, respectively) among all k-partitions. Given a set S of n objects, we want to find a k-partition of S which is farthest and most compact. It is generally hard to achieve the two goals simultaneously. In fact, the problem of finding a most compact k-partition with the smallest inner-cluster distance, even in the special case where the dissimilarities come from a metric space, is NP-hard (cf. [4]). There are, however, good cases where we can find such a k-partition rather easily. That is the case of a well-separated partition. A k-partition $\mathcal{P}$ of a set S is called well-separated if $D_{inn}(\mathcal{P}) < D_{int}(\mathcal{P})$. For this we can show the following:

Lemma 1 *Let S be a set of n objects such that dissimilarity is defined for every pair of objects. If S has a well-separated k-partition, then it is unique under the condition that no two pairs of objects have the same dissimilarity. The unique well-separated k-partition is farthest and also most compact.*

Next, we assume that it is known that there is a well-separated k-partition of S. If we sort all dissimilarities in increasing order then the inter-cluster distance must appear right next to the inner-cluster distance in the order. So, it suffices to find some dissimilarity t^* such that there is a well-separated k-partition with the inner-cluster distance being t^*. Because of a property of a well-separated k-partition, if we define a graph $G_t(S)$ by edges of weights at most t then the resulting connected components of the graph define a well-separated partition of S with the inner-cluster distance t. So, if we can find a dissimilarity t such that the graph $G_t(S)$ consists of k connected components, then it is a solution. In the following we sketch an algorithm for counting the number of connected components in the graph $G_t(S)$ in linear working space: We first scan every pair of objects and if their weight is at most t, we merge the two clusters containing those objects into one. Then we report the number of remaining clusters. After that, we again scan every pair of objects and report NO if we find a pair with dissimilarity greater than t such that both of them belong to the same cluster, and report YES if no such pair is found.

If S has a well-separated k-partition, the algorithm must return k and YES for dissimilarity δ_{ij} for some pair (i, j). A naive algorithm is to check all the dissimilarities. Since there are $O(n^2)$ different dissimilarities, $O(n^2)$ iterations of the algorithm are sufficient. This will require $O(n^4)$ time in total but the total space required is $O(n)$.

An idea for efficient implementation is to use binary search on the sorted list of dissimilarities. Generally speaking, as t value increases the number of subsets decreases. If the above algorithm outputs k and YES for some t^*, then the resulting partition is well-separated.

Linear-space algorithm for well-separated partition: One serious problem with the method sketched above is that we cannot store a sorted list of dissimilarities due to the linear space constraint. We implement the binary search in two stages. At the beginning our interval may contain a super linear number of distances. So, we compute an approximate median instead of the exact median. As the binary search proceeds, our interval gets shorter and shorter. Once the number of distances falling into the interval is at most cn for some positive constant c, then we can find an exact median. A more detailed description follows:

We start our binary search from the initial interval $[1, \binom{n}{2}]$ which corresponds to a distance interval determined by the smallest and largest distances, denoted by δ_1 and $\delta_{\binom{n}{2}}$, respectively. Generally, we maintain an index interval $[low, high]$ corresponding to the distance interval $[\delta_{low}, \delta_{high}]$, where δ_i denotes the i-th smallest distance. Imagine dividing the interval $[low, high]$ into 4 equal parts, then an approximate median is contained in the 2nd or 3rd quarters. Thus, half of the elements in $[low, high]$ are good for us. Equivalently, a random element is good with probability 1/2. How can we find one?

We pick a random integer k with $1 \leq k \leq high - low + 1$. We can evaluate the dissimilarity function in the order in which the dissimilarities are encountered when scanning the (unknown) distance matrix row by row to

simulate scanning the distance matrix. We refer to this process, which takes only $O(1)$ space, as *scanning the matrix*. We scan the matrix row by row and pick the k-th element X with $\delta_{low} \leq x \leq \delta_{high}$ that we encounter. Given X, we scan the matrix and count the number of values between δ_{low} and X, and also count the number of values between X and δ_{high}. In this way, we find out if X is an approximate median. If it is not, then we repeat the above. We know that the expected number of trials is 2. Assume that X is a good approximate median. While doing the above, we also find the index m such that $X = \delta_m$. Now we test if X is equal/larger/smaller than D_{inn}. If they are equal, we are done. Assume X is less than D_{inn}. Then, we set the right boundary *high* of our current interval to m. If X is larger than D_{inn}, then we set the left boundary *low* to m.

In this way, we spend $O(n^2)$ expected time for one binary search step. Since the expected number of these steps is $O(\log n)$, the overall expected time bound is $O(n^2 \log n)$. Once the current interval contains at most cn distances, we can apply an exact median finding algorithm although we have to scan the matrix in $O(n^2)$ time.

Theorem 2 *Given n objects, a function evaluating the dissimilarity between any pair of objects in $O(1)$ time, and an integer $k < n$, we can decide whether there is a well-separated k-partition or not in $O(n^2 \log n)$ expected time and $O(n)$ working space using an approximate median finding. Moreover, if there is such a partition, we can find it in the same time and space.*

Size Constrained Farthest k-partition

Let $e_1, e_2, \ldots, e_{n-1}$ be edges of a minimum spanning tree $MST(S)$, for a complete graph $G(S)$ defined for a set S of n objects, and assume that $|e_1| \leq |e_2| \leq \cdots \leq |e_{n-1}|$. Let $MST_k(S)$ be the set of components resulting after removing the $k-1$ longest edges $e_{n-1}, \ldots, e_{n-k-1}$ from $MST(S)$. Then, $MST_k(S)$ has exactly k components, which defines a k-partition of S and it is shown in [1] that this is a farthest k-partition of S. Moreover, $D_{int}(MST_k(S)) = |e_{n-k-1}|$.

Recall that our aim is to embed the graph using $O(n)$ space only. In order to use MDS for the embedding, clusters that are farthest are not sufficient. We also need to ensure that the clusters are sufficiently small and that there are not too many of them. Specifically, we need to find $O(\sqrt{n})$ clusters of size $O(\sqrt{n})$ each. Define *farthest partition satisfying the size constraint on c* as a clustering where all clusters contain less than $2c$ vertices and at most one cluster contains at most c vertices with $1 < c < n$ and $D_{int}(\mathcal{P})$ is maximized. The method in [1] does not provide such a partitioning.

To find the farthest partition of size $O(c)$ given a set S of n objects, consider the following algorithm. First, each object i is placed into a separate cluster C_i of size one to initialize the algorithm. The algorithm iteratively finds the minimum remaining dissimilarity δ_{ij}. If merging the cluster C_l containing object i and cluster C_m containing object j does not violate the size constraint, that is, if it does not produce a new cluster of size exceeding $2c$, the algorithm merges C_l and C_m into one cluster C_l. Dissimilarity δ_{ij} is then removed from consideration. These steps are iterated until all of the dissimilarities are removed from consideration.

Lemma 3 *Given c with $1 < c < n$, the algorithm creates a partition such that all clusters contain less than $2c$ vertices and at most one cluster contains at most c vertices. Furthermore, the partition is farthest among all partitions satisfying the size constraint.*

To find this partition, we need to find the minimum edge that has not yet been considered iteratively until all the edges were considered. The proposed algorithm is summarized below. In the algorithm we use a data structure for extracting edges in increasing order of their weights.

Algorithm Size-constrained_farthest_partition(k, c)
{for each $i = 1, \ldots, n$
  {$C_i := \{i\}$. Find an index $j > i$ such that
  $\delta_{i,j} := \min\{\delta_{i,l}, l = i+1, \ldots, n\}$.}
Build a list D to hold such minimum values along with indices.
$m := n$. // The number of clusters of sizes at most c.
while$(m > 1)$\{
  Take a record $(i, j, \delta_{i,j})$ that gives the minimum value δ_{ij} out of D.
  Scan the list $\delta_{i,i+1}, \delta_{i,i+2}, \ldots, \delta_{i,n}$ to find a minimum value greater than $\delta_{i,j}$, that is,
  $\delta_{i,j'} = \min\{\delta_{i,l} \geq \delta_{i,j}, l = i+1, \ldots, n, j \neq j'\}$.
  if there is such an element $\delta_{i,j'}$
    Insert a record $(i, j', \delta_{i,j'})$ in D.
  Find a cluster C_p that contains i by scanning the clusters.
  Find a cluster C_q that contains j similarly.
  if$(C_p \neq C_q$ and $|C_p| + |C_q| \leq 2c)$
    Merge C_q into C_p.
    if$(|C_p + C_q| > c)$ Decrement m.\}
Output remaining clusters.\}

To analyze the running time of the algorithm, note that the execution of the first for-loop takes $O(n^2)$ time. In the while-loop, there are at most $O(n^2)$ iterations and one execution of the while loop takes $O(n)$ time. Hence, the total running time is $O(n^3)$.

Theorem 4 *The algorithm runs in $O(n^3)$ time and uses $O(n)$ space.*

3 Graph Embedding

A direct way of embedding a weighted graph into a low-dimensional space is to apply LSMDS, which needs a full matrix representing dissimilarities between all pairs of objects. This takes $O(n^2)$ space for a set of n objects, which is often a problem for implementation. To remedy this, we partition the given set into $O(\sqrt{n})$ clusters of size $O(\sqrt{n})$ each by applying the algorithm in the previous section. Suppose we have $k = O(\sqrt{n})$ clusters $C_1, C_2, \ldots, C_k$ with $|C_i| = O(\sqrt{n})$ for each i.

First, find a center object in each cluster. A *center object* in a cluster C_i, denoted by $center(C_i)$, is defined as an object in C_i such that the largest distance to any other object in that cluster is smallest. We denote the i-th cluster by $C_i = \{p_i = p_{i_1}, p_{i_2}, \ldots, p_{i_{n_i}}\}$ with the first element $p_i = p_{i_1}$ as its cluster center.

Second, we form a set $C_0 = \{p_1, p_2, \ldots, p_k\}$ consisting of cluster centers. Since $k = O(\sqrt{n})$, we can apply LSMDS to find an optimal embedding of elements in C_0 using a distance matrix of size $O(n)$. We fix those points.

Third, we embed clusters one by one. We apply LSMDS to the cluster C_i to have a set $P(C_i)$ of points reflecting dissimilarities among C_i. Note that we still have the freedom to rotate and flip the points in $P(C_i)$ around the cluster center p_i. We compute an optimal angle θ for a rotation such that the total error between distances and dissimilarities of points in C_i and the cluster centers excluding p_i is minimized. More precisely, we want to minimize $\sum_{p_{i_j}, p_l (j=1,\cdots,n_i, l=1,\cdots,k(l\neq i))} [d(p_{i_j}(\theta), p_l) - \delta(p_{i_j}, p_l)]^2$, where $p_{i_j}(\theta)$ is a point after rotating $p_{i_j} = (x_{i_j}, y_{i_j})$ by θ around the cluster center p_i of C_i in the clockwise direction. Note that this expression is similar to E_{LSMDS} (Equation 2). Once we find the optimal angles for the original placement and the flipped placement of points, we choose the configuration that yields the minimum. Unfortunately, it seems hard to find such an optimal angle θ. So, we relax the condition as follows: $f(\theta) = \sum_{p_{i_j}, p_l} d(p_{i_j}(\theta), p_l)^2 - \delta(p_{i_j}, p_l)^2 = \sum_{j=1}^{n_i} \sum_{l=1}^{k} (x_{i_j} \cos\theta + y_{i_j} \sin\theta - x_l)^2 + (y_{i_j} \cos\theta - x_{i_j} \sin\theta - y_l)^2 - \delta_{i_j,l}^2 = \sum_{j=1}^{n_i} \sum_{l=1}^{k} x_{i_j}^2 + y_{i_j}^2 + x_l^2 + y_l^2 - 2x_{i_j}x_l \cos\theta - 2y_{i_j}x_l \sin\theta + 2x_{i_j}y_l \sin\theta - 2y_{i_j}y_l \cos\theta - \delta_{i_j,l}^2$. Differentiating $f(\theta)$ by θ and setting it to 0, we obtain $\tan\theta = \frac{\sum_l \sum_j y_{i_j}x_l - x_{i_j}y_l}{\sum_l \sum_j x_{i_j}x_l + y_{i_j}y_l}$. Here, (x_l, y_l) are the coordinates of the point p_l and (x_{i_j}, y_{i_j}) are that of p_{i_j}.

Theorem 5 *There exists an algorithm to embed a data set S in the plane while approximating pairwise dissimilarities between objects in S using $O(n^3)$ time and $O(n)$ space.*

References

[1] T. Asano, B. Bhattacharya, M. Keil, and F. Yao: Clustering algorithms based on minimum and maximum spanning trees. *4 SoCG*:252 - 257, 1988.

[2] A. M. Bronstein, M. M. Bronstein, R. Kimmel: Three-dimensional face recognition. *Int. Jl. Comp. Vision 64*(1):5–30, 2005.

[3] T. Cox, and M. Cox: Multidimensional Scaling. *Chapman & Hall CRC*, 2001.

[4] T. Gonzalez: Clustering to minimize the maximum intercluster distance. In *Theoretical Comput. Sci.*, 38, pp:293-306, 1985.

[5] P. Groenen and P.H. Franses: Visualizing time-varying correlations across stock markets. In *Jl. Empirical Finance*, 7, 155-172, 2000.

[6] G. Zigelman, R. Kimmel, N. Kiryati: Texture mapping using surface flattening via multi-dimensional scaling. *IEEE Trans. Vis. & Comp. Graphics 8*(2):198–207, 2002.