Nonlinear Manifold Learning For Data Stream

Martin H. C. Law* Nan Zhang* Anil K. Jain*

Abstract

There has been a renewed interest in understanding the structure of high dimensional data set based on *manifold learning*. Examples include ISOMAP [25], LLE [20] and Laplacian Eigenmap [2] algorithms. Most of these algorithms operate in a "batch" mode and cannot be applied efficiently for a data stream. We propose an incremental version of ISOMAP. Our experiments not only demonstrate the accuracy and efficiency of the proposed algorithm, but also reveal interesting behavior of the ISOMAP as the size of available data increases.

1 Introduction

Data mining usually involves understanding the structure of large high dimensional data sets. Typically, the underlying structure of the data is assumed to be on a hyperplane. This assumption can be too restrictive when the data points actualy lie on a nonlinear manifold. A knowledge of the manifold can help us to transform the data to a low-dimensional space with little loss of information, enabling us to visualize data, as well as performing classification and clustering more efficiently. A separate issue in data mining is that sometimes information is collected sequentially through a data stream. In such situations, it would be very helpful if we can update our analysis using the additional data points that become available. Thus, the goal of this paper is to investigate how we can recover a nonlinear manifold given a data stream.

One of the earliest nonlinear dimensionality reduction techniques is the Sammon's mapping [22]. Over time, other nonlinear methods have been proposed, such as self organizing maps (SOM) [16], principal curve and its extensions [13, 26], auto-encoder neural networks [1, 10], generative topographic maps (GTM) [4] and kernel principal component analysis (KPCA) [23]. A comparison of some of these methods can be found in [17]. Many of these algorithms learn a mapping from the high dimensional space to a low dimensional space explicitly. An alternative approach is based on the notion of manifold that has received considerable attention recently. Representative techniques of this approach include isometric feature mapping (ISOMAP) [25], which estimates the geodesic distances on the manifold and uses them for projection, local linear embedding (LLE) [20], which projects points to a low dimensional space that preserves local geometric properties, and Laplacian Eigenmap [2], which can be viewed as finding the coefficients of a set of smooth basis functions on the manifold. One can also model a manifold by a mixture of Gaussians and recover the global co-ordinates by combining the co-ordinates from different Gaussian components [5, 21, 24, 27], or by other methods [28]. A related problem in manifold learning is to estimate the intrinsic dimensionality of the manifold. Different algorithms have been considered [19, 14].

Most of these algorithms operate in a batch mode[1], meaning that all data points need to be available during training. When data points arrive sequentially, batch methods are computationally demanding: repeatedly running the "batch" version whenever new data points are obtained takes a long time. Data accumulation is particularly beneficial to manifold learning algorithms, because many of them require a large amount of data in order to satisfactorily learn the manifold. Another desirable feature of incremental methods is that we can visualize the evolution of the data manifold. As more and more data points are obtained, visualization of the change in the manifold may reveal some interesting properties of the data stream. In our experiments, we have composed a AVI video clip[2] to show how the manifold changes as we transit from a small to a large data set.

Adaptiveness is also an advantage of incremental manifold learning – the algorithm can adjust the manifold in the presence of gradual changes. For example, suppose we learn the manifold of the face images of N individuals in order to improve the performance of face recognition system. Over time, faces of different people change gradually. This is referred as the aging effect, one of the most challenging issues in face recognition. The system performance can be improved if the manifold of face images can be adjusted according to these

*Dept. of Comp. Sci. and Eng., Michigan State University, East Lansing, MI 48823, USA

[1]Note that Sammon's mapping can be implemented by a feed-forward neural network [17] and hence can be made online if we use an online training rule.

[2]`http://www.cse.msu.edu/~lawhiu/iisomap.html`

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited

facial changes.

In this paper, we have modified the ISOMAP algorithm to use data stream as the input. We have decided to focus on the ISOMAP algorithm because it is intuitive, well understood and produces reasonable mapping results [15, 31]. Also, there are theoretical studies supporting the use of ISOMAP, such as its convergence proof [3] and when it can recover the co-ordinates [11]. There is also a continuum extension of ISOMAP [32].

The main contributions of this study are:

1. An incremental geodesic distance updating rule. The geodesic distance is used in ISOMAP.

2. Methods to incrementally update the topological co-ordinates. The proposed methods are independent of the definition of the geodesic structure, so they could also be used in other incremental nonlinear dimension reduction methods.

3. A method to visualize the data manifold to interpret changes in the data stream.

The rest of this paper is organized as follows. After a recap of ISOMAP in section 2, the proposed incremental methods are described in section 3. Experimental results are presented in section 4, followed by discussion in section 5. Finally, in section 6 we conclude and describe some topics for future work.

2 ISOMAP

Given a set of data points $\mathbf{y}_1, \ldots, \mathbf{y}_n$ in a high dimensional space, ISOMAP assumes that the data lie on a manifold of dimension d and tries to find the global co-ordinates of those points on the manifold. Let $\mathbf{x}_i \in \mathcal{R}^d$ be the co-ordinates corresponding to $\mathbf{y}_i$.[3] ISOMAP attempts to recover an isometric mapping from the co-ordinate space to the manifold. One may view $\mathbf{x}_i$ as the (nonlinearly) reduced dimension representation of $\mathbf{y}_i$. Define $\mathbf{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_n)$. Let Δ_{ij} be the distance between $\mathbf{y}_i$ and $\mathbf{y}_j$. ISOMAP also requires the user to specify the neighborhood. It can either be ϵ-neighborhood, where $\mathbf{y}_i$ and $\mathbf{y}_j$ are neighbors if Δ_{ij} is less than a parameter ϵ, or knn-neighborhood, where $\mathbf{y}_i$ and $\mathbf{y}_j$ are neighbors if $\mathbf{y}_i$ ($\mathbf{y}_j$) is one of the k nearest neighbors (knn) of $\mathbf{y}_j$ ($\mathbf{y}_i$). The value of k is specified by the user.

The ISOMAP algorithm first constructs a weighted undirected neighborhood graph $\mathcal{G} = (V, E)$ with the vertex $v_i \in V$ corresponding to $\mathbf{y}_i$. An edge between v_i and v_j, $e(i,j)$, exists iff $\mathbf{y}_i$ is a neighbor of $\mathbf{y}_j$. The weight of $e(i,j)$, w_{ij}, is simply Δ_{ij}. Let g_{ij} denote the length of the shortest path $sp(i,j)$ between v_i and v_j. The shortest paths can be found by the Floyd-Warshall algorithm or the Dijkstra's algorithm with different source vertices [7], and the shortest paths can be stored efficiently by the predecessor matrix π_{ij}, where $\pi_{ij} = k$ if v_k is immediately before v_j in $sp(i,j)$. Since g_{ij} can be regarded as the approximate geodesic distance between $\mathbf{y}_i$ and $\mathbf{y}_j$, we shall call g_{ij} "geodesic distance". Note that $\mathbf{G} = \{g_{ij}\}$ is a symmetric matrix. By assuming $\sum_i \mathbf{x}_i = 0$, the target inner product matrix $\mathbf{B}$ can be found by $\mathbf{B} = \mathbf{HGH}$, where $\mathbf{H} = \{h_{ij}\}$, $h_{ij} = \delta_{ij} - 1/n$ and δ_{ij} is the delta function, i.e., $\delta_{ij} = 1$ if $i = j$ and 0 otherwise. We seek $\mathbf{X}^T\mathbf{X}$ to be as close to $\mathbf{B}$ as possible by setting $\mathbf{X} = (\sqrt{\lambda_1}\mathbf{v}_1 \ \ldots \sqrt{\lambda_d}\mathbf{v}_d)^T$, where $\lambda_1, \ldots, \lambda_d$ are the d largest eigenvalues of $\mathbf{B}$, with corresponding eigenvectors $\mathbf{v}_1, \ldots, \mathbf{v}_d$. Note that computing $\mathbf{HGH}$ is effectively a centering operation on $\mathbf{G}$, and this can be computed in $O(n^2)$ time.

3 Incremental Version of ISOMAP

Suppose we have the co-ordinates $\mathbf{x}_i$ of $\mathbf{y}_i$ for $1 \leq i \leq n$. The new sample $\mathbf{y}_{n+1}$ arrives and the goal of incremental ISOMAP is to update the co-ordinates $\mathbf{x}_i$ so as to best preserve the updated geodesic distances. This is done in three stages. We first update g_{ij} for the original n vertices. The points $\mathbf{x}_1, \ldots, \mathbf{x}_n$ are then updated because of the changes in g_{ij}. Finally, $\mathbf{x}_{n+1}$, the co-ordinate of the new sample, is found. Proofs and details of the algorithms are described in the Appendix.

3.1 Updating the Geodesic Distances The point $\mathbf{y}_{n+1}$ introduces a new vertex v_{n+1} in the graph $\mathcal{G}$. At first sight, it seems straightforward to incorporate the influence of v_{n+1} on the geodesic distances, but the new vertex can change the neighborhood structure and break an edge in an existing shortest path, as well as creating an improved shortest path.

Appendices A and C describe our algorithm in details for updating the geodesic distances. The basic idea is that we first find the set of edges that need to be removed or added because of v_{n+1}. For each edge $e(a,b)$ that needs to be removed, we "propagate" from v_a and v_b to find all (i,j) pairs such that the shortest path from v_i to v_j uses $e(a,b)$. The geodesic distances of these vertex pairs need to be re-computed, and this is done by a modified version of Dijkstra's algorithm. The added edges, which are incident on v_{n+1}, may create a better shortest path. We check the neighbors of v_{n+1} to see if this happens or not. If yes, the effect of the better shortest path is also propagated to other vertices.

[3] In the original ISOMAP paper [25], the i-th data point is simply denoted by i, and $\mathbf{y}_i$ is used to denote the embedded co-ordinate of i. In this paper, we instead adopt the notation used in [8].

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited

While the proposed algorithm is applicable for both *knn* and ϵ neighborhoods, we shall focus on the *knn* neighborhood as it is more suitable for incremental learning. During the incremental learning, the graph can be temporarily disconnected. A simple solution is to embed the largest graph component first, and then add back the excluded vertices when they become connected again as more data points become available.

3.2 Updating the Co-ordinates We need to update the co-ordinates based on the modified geodesic distance matrix $\mathbf{G}_{\text{new}}$. One may view this as an incremental eigenvalue problem, as the co-ordinates $\mathbf{x}_i$ can be obtained by eigen-decomposition. However, since the size of the geodesic distance matrix is increasing, traditional methods (such as those described in [30] or [6]) cannot be applied directly. We propose to use two common iterative updating schemes.

Let $\mathbf{G}_{\text{new}}$ denote the matrix of updated geodesic distances. Given $\mathbf{B} = \mathbf{HGH}$ and $\mathbf{X}$ such that $\mathbf{B} \approx \mathbf{X}^T\mathbf{X}$, our goal is to find the new $\mathbf{X}_{\text{new}}$ such that $\mathbf{X}_{\text{new}}^T\mathbf{X}_{\text{new}} \approx \mathbf{B}_{\text{new}}$, where $\mathbf{B}_{\text{new}} = \mathbf{H}\mathbf{G}_{\text{new}}\mathbf{H}$. Our first approach is based on gradient descent. The eigen decomposition in batch ISOMAP is equivalent to finding $\mathbf{X}$ that minimizes

$$J(\mathbf{B}, \mathbf{X}) = tr\big((\mathbf{B} - \mathbf{X}^T\mathbf{X})(\mathbf{B} - \mathbf{X}^T\mathbf{X})^T\big)/n^2, \tag{3.1}$$

which is the average of the square of the entries in $\mathbf{B} - \mathbf{X}\mathbf{X}^T$. Its gradient is

$$\nabla_{\mathbf{X}} J(\mathbf{B}, \mathbf{X}) = (-4\mathbf{XB} + 4\mathbf{X}\mathbf{X}^T\mathbf{X})/n^2, \tag{3.2}$$

and we update the co-ordinates[4] by $\mathbf{X}_{\text{new}} = \mathbf{X} - \alpha\nabla_{\mathbf{X}} J(\mathbf{X}, \mathbf{B}_{\text{new}})$. While there exist many schemes to select the step size α, we empirically set its value to $\alpha = 0.003$. This approach is fast (we descent only once) and $\mathbf{X}$ is changed smoothly, thereby leading to a good visualization.

Another approach to update $\mathbf{X}$ is to find the eigenvalues and eigenvectors of $\mathbf{B}_{\text{new}}$ by an iterative approach. We first recover (approximately) the eigenvectors of $\mathbf{B}$ from $\mathbf{X}$ by normalizing the i-th column of $\mathbf{X}^T$ to norm one to obtain the i-th eigenvector $\mathbf{v}_i$ and form $\mathbf{V} = (\mathbf{v}_1, \ldots, \mathbf{v}_d)$ as a reasonable initial guess of the eigenvectors of $\mathbf{B}_{\text{new}}$. Subspace iteration together with Rayleigh-Ritz acceleration [12] is used to refine $\mathbf{V}$ as eigenvectors of $\mathbf{B}_{\text{new}}$:

1. Compute $\mathbf{Z} = \mathbf{B}_{\text{new}}\mathbf{V}$ and perform QR decomposition on $\mathbf{Z}$, i.e., we write $\mathbf{Z} = \mathbf{QR}$ and let $\mathbf{V} = \mathbf{Q}$.

2. Form $\mathbf{Z} = \mathbf{V}^T\mathbf{B}_{\text{new}}\mathbf{V}$ and perform eigen-decomposition of the d by d matrix $\mathbf{Z}$. Let λ_i and $\mathbf{u}_i$ be the i-th eigenvalue and the corresponding eigenvector.

3. $\mathbf{V}_{\text{new}} = \mathbf{V}[\mathbf{u}_1 \ldots \mathbf{u}_d]$ is the improved set of eigenvectors of $\mathbf{B}_{\text{new}}$.

Since d is small (typically 2 or 3 for visualization purposes), the time for eigen-decomposition of $\mathbf{Z}$ is negligible. We do not use any variant of inverse iteration because $\mathbf{B}_{\text{new}}$ is not sparse and its inversion takes $O(n^3)$ time.

3.2.1 Finding the Co-ordinates of the New Sample $\mathbf{x}_{n+1}$ is found by matching its inner product with $\mathbf{x}_i$ to be as close to the target value as possible. Let $\gamma_i = ||\mathbf{x}_i - \mathbf{x}_{n+1}||^2$. Since $\sum_{i=1}^n \mathbf{x}_i = \mathbf{0}$, it is easy to show that

$$\begin{aligned} ||\mathbf{x}_{n+1}||^2 &= \frac{1}{n}\Big(\sum_{i=1}^n \gamma_i - \sum_{i=1}^n ||\mathbf{x}_i||^2\Big) \\ \text{and} \quad \mathbf{x}_{n+1}^T\mathbf{x}_i &= -\frac{1}{2}\big(\gamma_i - ||\mathbf{x}_{n+1}||^2 - ||\mathbf{x}_i||^2\big) \quad \forall i \end{aligned} \tag{3.3}$$

By replacing γ_i with the actual geodesic distance $g_{i,n+1}$, we obtain our target inner product between $\mathbf{x}_{n+1}$ and $\mathbf{x}_i$, f_i, in a manner similar to equation (3.3). $\mathbf{x}_{n+1}$ can be found by solving (in least-square sense) the equation $\mathbf{X}^T\mathbf{x}_{n+1} = \mathbf{f}$, where $\mathbf{f} = (f_1, \ldots, f_n)^T$. Alternatively, we can initialize $\mathbf{x}_{n+1}$ randomly and then apply an iterative method to refine its value. However, this is not a good idea, since the co-ordinate of the newly arrived data can be obtained in a straightforward manner as above, and the user is usually interested in a good estimate of the co-ordinate of the new data point.

After obtaining the new $\mathbf{x}_{n+1}$, we normalize them so that the center of all the $\mathbf{x}_i$ is at the origin.

3.3 Complexity In appendix E, we show that the overall complexity of the geodesic distance update can be written as $O(q(|F| + |H|))$, where F and H contain vertex pairs whose geodesic distances are lengthened and shortened because of v_{n+1}, respectively. We also want to point out that algorithm 3 in appendix C is reasonably efficient; its complexity to solve the all-pair shortest path by forcing all geodesic distances to be updated is $O(n^2 logn + n^2 q)$. This is the same as the complexity of the best known algorithm for the all-pair shortest path problem of a sparse graph, which involves running Dijkstra's algorithm multiple times with different source vertices.

For the update of co-ordinates, both gradient descent and subspace iteration for co-ordinate update take $O(n^2)$ time because of the matrix multiplication. We

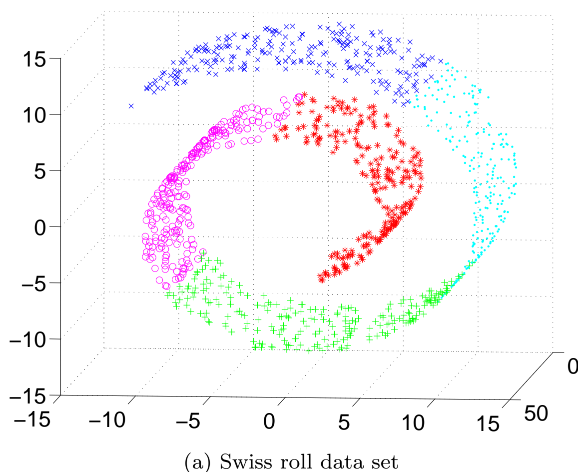
[4] Although $J(\mathbf{B}, \mathbf{X})$ can have many saddle points with $\mathbf{X}$ consisting of eigenvectors of $\mathbf{B}$, this does not seem to affect the gradient descent algorithm in practice.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited

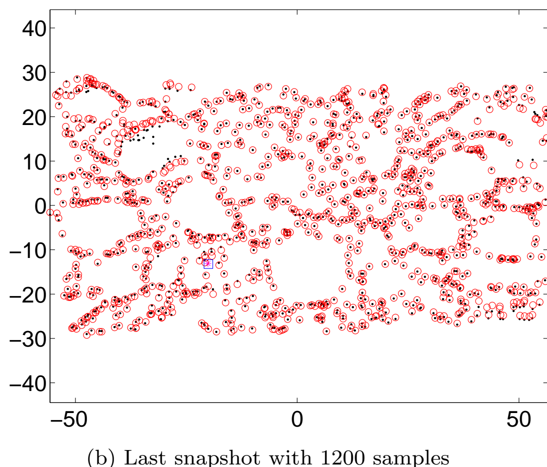
are exploring different methods that make use of the sparseness of the change in the geodesic distance matrix in order to reduce its complexity. Section 6 also describes other alternatives to cope with this issue.

4 Experiments

Our first experiment is on the Swiss roll data set (Fig. 1(a)), which is also used in the original ISOMAP paper. We use the *knn* neighborhood with $k = 5$. We first learn an initial manifold of 30 samples by the batch ISOMAP. The data points are then added in a random order using the proposed incremental ISOMAP until we get a total of 1200 samples. Fig. 1(b) shows the result. Circles and dots represent the sample co-ordinates in the manifold computed by the batch ISOMAP and the incremental ISOMAP, respectively. We can see that the final result of the incremental ISOMAP is almost the same as the batch version. The video clip at `http://www.cse.msu.edu/~lawhiu/manifold/iisomap.html` shows the results of the intermediate stages as the data points arrive. At first, the co-ordinates computed by the incremental ISOMAP are far away from the target values because the shortest path distances do not estimate the geodesic distances on the manifold accurately. As additional data points arrive, the shortest path distances become more reliable and the co-ordinates of the incremental ISOMAP converge to those computed by batch ISOMAP.



(a) Swiss roll data set



(b) Last snapshot with 1200 samples

Figure 1: Incremental ISOMAP on Swiss roll data set. The original data points are shown in (a). In (b), the circles (o) and the dots (.) correspond to the target and estimated co-ordinates, respectively.

4.1 Global Rearrangement of Co-ordinates

During our experiments, we were surprised that the co-ordinates sometimes can change dramatically after adding just a *single* sample (Fig. 2). The addition of a new sample can delete critical edges in the graph and this can change the geodesic distances dramatically. Fig. 2(c) explains why: when the "short-circuit" edge e is deleted, the shortest path from any vertex in A to any vertex in B becomes much longer. This leads to a substantial change of the geodesic distances and hence the co-ordinates.

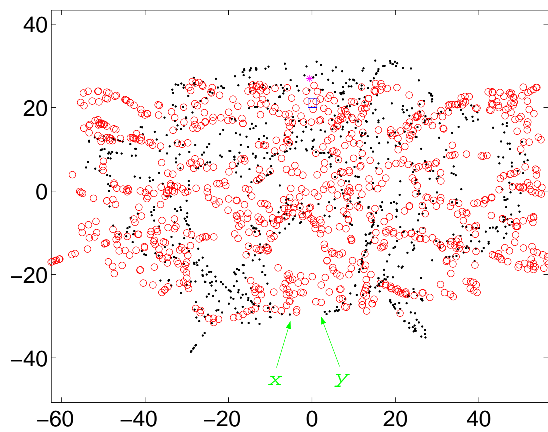
4.2 Approximation Error and Computation Time

Because the geodesic distances are exactly updated, the only approximation error in the incremental ISOMAP arises from the co-ordinate update. The error can be estimated by comparing the co-ordinates from our updating schemes with the co-ordinates from the exact eigen-solver (Fig. 3). When there is a major change in geodesic distances, the error increases sharply. It then dies down quickly when more samples come. Both methods converge to the target co-ordinates, with subspace iteration showing higher accuracy.

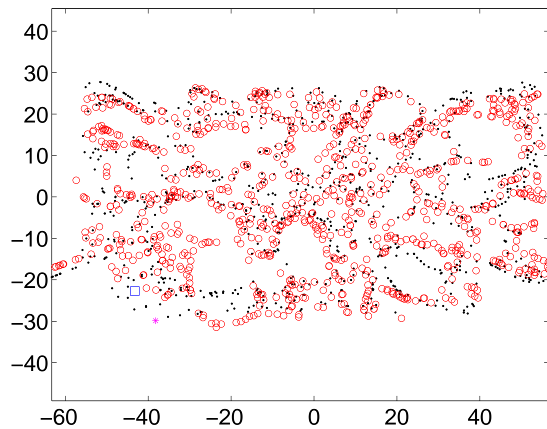
Regarding computation time, we note that most of the computation involves updating the geodesic distances based on the set of removed and inserted edges, and updating the co-ordinates based on the new geodesic distance matrix. We measure the running time of our algorithm on a Pentium IV 1.8 GHz PC with 512MB memory. We have implemented our algorithm mostly in Matlab, though the graph algorithms are written in C. The times for gradient descent, subspace iteration and the exact eigen-solver are 14.9s, 48.6s and 625.5s, respectively[5]. Both gradient descent and subspace iteration are more efficient than the exact solver. The gradient descent method is faster because it involves only one matrix multiplication. For the update of geodesic distances, our algorithm takes 82s altogether. If we run the C implementation of Dijkstra's algorithm

[5]Note that all these operations are performed in Matlab and hence their comparison is fair.

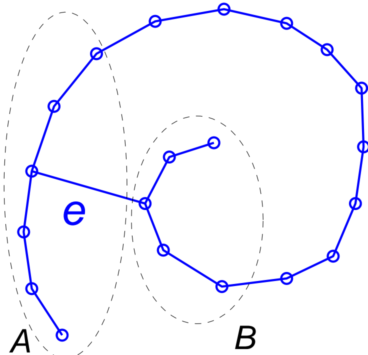
Copyright © by SIAM. Unauthorized reproduction of this article is prohibited



(a) 903 samples

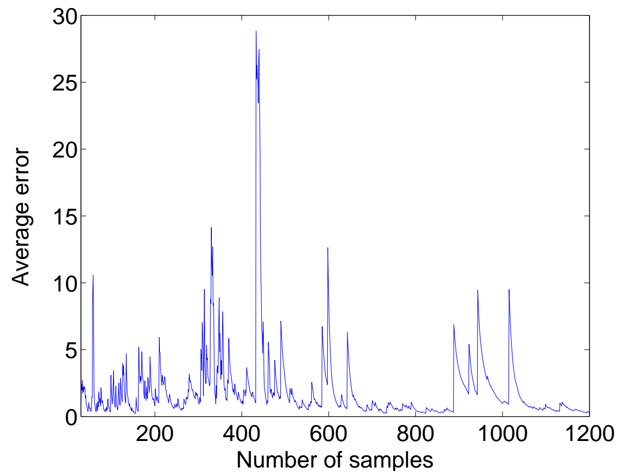


(b) 904 samples

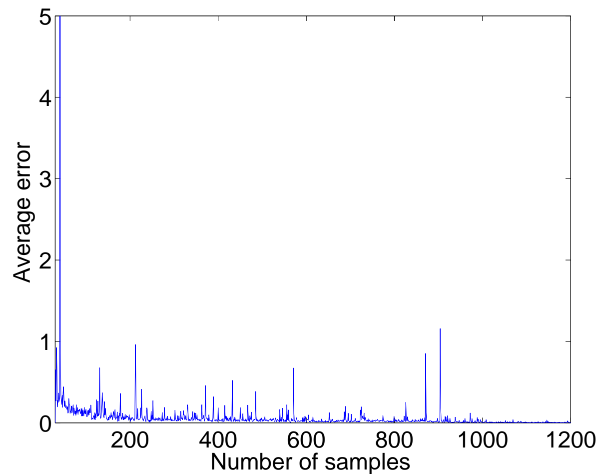


(c)

Figure 2: A single sample can change the co-ordinates dramatically. The addition of the 904-th sample breaks an edge connecting x and y in (a), leading to a "flattening" of the co-ordinates as shown in (b). (c) explains why the geodesic distances can change dramatically.



(a) Gradient descent



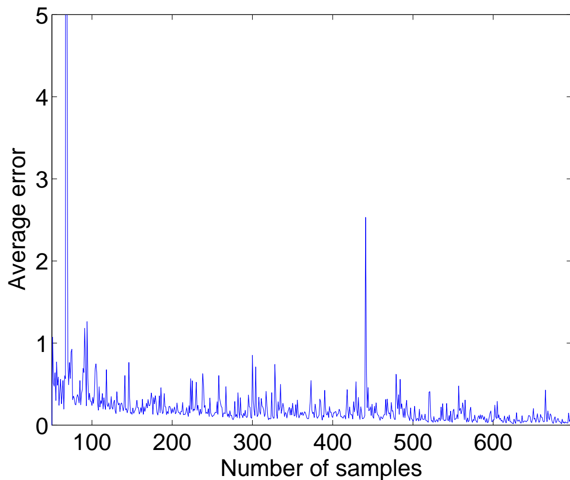
(b) Subspace iteration

Figure 3: L_1 distance between the co-ordinates obtained from the proposed updating methods and the exact eigen-solver. Typical values of the co-ordinates can be seen in figure 2. The co-ordinates are first aligned to have diagonal covariance matrices and the same order of variances before the error is computed.

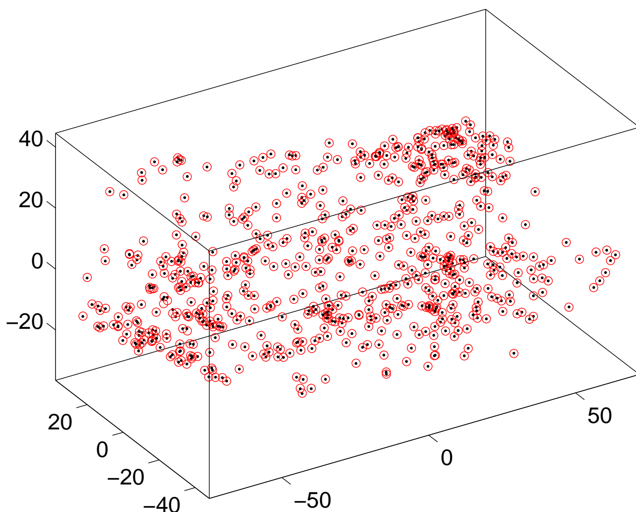
Copyright © by SIAM. Unauthorized reproduction of this article is prohibited

in [25] repeatedly, it takes 1457s. This shows that our algorithm is indeed more efficient for updating both the geodesic distances and the co-ordinates.

4.3 The Face Image Data Set We also tested our incremental ISOMAP on the face image data available at the ISOMAP website `http://isomap.stanford.edu`. This data set consists of 698 synthesized face images (64 by 64 pixels) in different poses and lighting conditions. The intrinsic dimensionality of the manifold is 3. The average error and the final snapshot are shown in Fig. 4. We can see that our algorithm, once again, estimates the co-ordinates accurately.



(a) Approximation error



(b) Last snapshot (698 data points)

Figure 4: Results of incremental ISOMAP on the face data set. Circles and dots correspond to the target and estimated co-ordinates, respectively.

5 Discussion

Our algorithm is reasonably general and can be applied in other common online learning settings. For example, if we want to extend our algorithm to delete samples collected in distant past, we simply need to change the set D in Appendix B to be the set of edges incident on the samples to be deleted, and then execute the algorithm. Another scenario is that some of the existing $\mathbf{y}_i$ are altered, possibly due to the change of the environment. In this case, we first calculate the new weights of the edges in a straightforward manner. If the weight of an edge increases, we modify the algorithm in Appendix B in order to update the geodesic distances, as edge deletion is just a special case of weight increase. On the other hand, if the edge weight decreases, algorithm 5 can be used. We then update the co-ordinates based on the change in geodesic distance as described in section 3.2.

As far as convergence is concerned, the output of the incremental ISOMAP can be made identical to that of the batch ISOMAP if the gradient descent or the subspace iteration is run repeatedly for each new sample. Obviously, this is computationally unattractive. The fact that we execute gradient descent or subspace iteration only once can be regarded as a tradeoff between theoretical convergence and practical efficiency, though the convergence is excellent in practice. The dimension, d, of $\mathbf{x}_i$, can be estimated from the data by examining the residue of $\mathbf{B} - \mathbf{X}^T\mathbf{X}$ in a manner similar to [25].

6 Conclusions and Future Work

We have presented an incremental version of the ISOMAP algorithm. We have solved the graph theory problem of updating the geodesic distances and the numerical problem of updating the co-ordinates. Our experiments demonstrate the efficiency and accuracy of the proposed method.

There are several directions for future work. The current algorithm is not fully online, because ISOMAP is a global algorithm: for any sample, we need to consider how it interacts with the other samples before we can find its co-ordinate. There are several possible ways to tackle this. The simplest approach is to discard the oldest sample when we have accumulated a sufficient number of samples. This also has the additional benefit of making the algorithm adaptive. Alternatively, we can maintain a set of “landmark points” [9] of constant size and consider the relationship of the new sample with only the landmark points. Finally, we can compress the data by, say, Gaussians that lie along the manifold [29].

We can improve the efficiency of co-ordinate update by making use of the sparseness of the change in geodesic distances. Non-exact but possibly more effi-

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited

cient algorithms for updating the geodesic distances can be considered. Ideas similar to the distance vector or link state in network routing are worthy of investigation. We can also consider the online version of other manifold learning algorithms, using the tools proposed as building blocks.

Acknowledgement

This research was supported by ONR contract # N00014-01-1-0266.

References

[1] P. Baldi and K. Hornik. Neural networks and principal component analysis: learning from examples without local minima. *Neural Networks*, 2:53–58, 1989.

[2] M. Belkin and P. Niyogi. Laplacian eigenmaps and spectral techniques for embedding and clustering. In *Advances in Neural Information Processing Systems 14*, pages 585–591. MIT Press, 2002.

[3] M. Bernstein, V. de Silva, J. Langford, and J. Tenenbaum. Graph approximations to geodesics on embedded manifolds. Technical report, Department of Psychology, Stanford University, 2000.

[4] C. M. Bishop, M. Svensen, and C. K. I. Williams. GTM: the generative topographic mapping. *Neural Computation*, 10:215–234, 1998.

[5] M. Brand. Charting a manifold. In *Advances in Neural Information Processing Systems 15*, pages 961–968. MIT Press, 2003.

[6] M. Brand. Fast online SVD revisions for lightweight recommender systems. In *Proc. SIAM International Conference on Data Mining*, 2003. `http://www.siam.org/meetings/sdm03/proceedings/sdm03_04.pdf`.

[7] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[8] T. F. Cox and M. A. A. Cox. *Multidimensional Scaling*. Chapman & Hall, 2001.

[9] V. de Silva and J. B. Tenenbaum. Global versus local approaches to nonlinear dimensionality reduction. In *Advances in Neural Information Processing Systems 15*, pages 705–712. MIT Press, 2003.

[10] D. DeMers and G. Cottrell. Non-linear dimensionality reduction. In *Advances in Neural Information Processing Systems*, volume 5, pages 580–587. Morgan Kaufmann, 1993.

[11] D. L. Donoho and C. Grimes. When does isomap recover natural parameterization of families of articulated images? Technical Report 2002-27, Department of Statistics, Stanford University, August 2002.

[12] G. H. Golub and C. F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, 1996.

[13] T. Hastie and W. Stuetzle. Principal curves. *Journal of the American Statistical Association*, 84:502–516, 1989.

[14] D. R. Hundley and M. J. Kirby. Estimation of topological dimension. In *Proc. SIAM International Conference on Data Mining*, 2003. `http://www.siam.org/meetings/sdm03/proceedings/sdm03_18.pdf`.

[15] O. C. Jenkins and M. J Mataric. Automated derivation of behavior vocabularies for autonomous humanoid motion. In *Proc. of the Second Int'l Joint Conference on Autonomous Agents and Multiagent Systems*, Melbourne, Austrailia, July 2003.

[16] T. Kohonen. *Self-Organizing Maps*. Springer-Verlag, 2001. 3rd edition.

[17] J. Mao and A. K. Jain. Artificial neural networks for feature extraction and multivariate data projection.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited

IEEE Transactions of Neural Networks, 6(2):296–317, March 1995.

[18] E. M. Palmer. *Graph Evolution: An Introduction to the Theory of Random Graphs.* John Wiley & Sons, 1985.

[19] K. Pettis, T. Bailey, A. K. Jain, and R. Dubes. An intrinsic dimensionality estimator from near-neighbor information. *IEEE Transactions of Pattern Analysis and Machine Intelligence*, 1(1):25–36, January 1979.

[20] S. T. Roweis and L. K. Saul. Nonlinear dimensionality reduction by locally linear embedding. *Science*, 290:2323–2326, 2000.

[21] S. T. Roweis, L. K. Saul, and G. E. Hinton. Global coordination of local linear models. In *Advances in Neural Information Processing Systems 14*, pages 889–896. MIT Press, 2002.

[22] J. W. Sammon. A non-linear mapping for data structure analysis. *IEEE Transactions on Computers*, C-18(5):401–409, May 1969.

[23] B. Schölkopf, A.J. Smola, and K.-R. Müller. Nonlinear component analysis as a kernel eigenvalue problem. *Neural Computation*, 10:1299–1319, 1998.

[24] Y. W. Teh and S. T. Roweis. Automatic alignment of local representations. In *Advances in Neural Information Processing Systems 15*, pages 841–848. MIT Press, 2003.

[25] J.B. Tenenbaum, V. de Silva, and J.C. Langford. A global geometric framework for nonlinear dimensionality reduction. *Science*, 290:2319–2323, 2000.

[26] R. Tibshirani. Principal curves revisited. *Statistics and Computing*, 2:183–190, 1992.

[27] J.J. Verbeek, N. Vlassis, and B. Krose. Coordinating principal component analyzers. In *Proceedings of International Conference on Artificial Neural Networks*, pages 914–919, Madrid, Spain, 2002.

[28] J.J. Verbeek, N. Vlassis, and B. Krose. Fast nonlinear dimensionality reduction with topology preserving networks. In *Proc. 10th European Symposium on Artificial Neural Networks*, pages 193–198, 2002.

[29] P. Vincent and Y. Bengio. Manifold parzen windows. In *Advances in Neural Information Processing Systems 15*, pages 825–832. MIT Press, 2003.

[30] J. Weng, Y. Zhang, and W.S. Hwang. Candid covariance-free incremental principal component analysis. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 25(8):1034–1040, 2003.

[31] M.-H. Yang. Face recognition using extended isomap. In *International Conference on Image Processing*, pages II: 117–120, 2002.

[32] H. Zha and Z. Zhang. Isometric embedding and continuum isomap. In *International Conference on Machine Learning*, 2003. `http://www.hpl.hp.com/conferences/icml2003/papers/8.pdf`.

Appendix A Update of Neighborhood Graph

The addition of v_{n+1} modifies the neighborhood graph. Let A and D denote the set of edges that are added and deleted from the graph, respectively.

For ϵ-neighborhood, there is an edge between v_i and v_{n+1} iff $\Delta_{i,n+1} \le \epsilon$. Also, Δ_{ij} is not affected by v_{n+1} for $1 \le i \le n$ and $1 \le j \le n$. So no edge is deleted. Therefore,

$$\begin{aligned} A &= \{e(i, n+1) : i \in 1, \ldots, n \text{ and } \Delta_{i,n+1} \le \epsilon\} \\ D &= \emptyset. \end{aligned} \tag{A.1}$$

The case for *knn*-neighborhood is more complicated. $e(i, n+1)$ is added if v_i is one of the *knn* of v_{n+1}, or v_{n+1} is one of the *knn* of v_i. Let τ_i be the index of the k-th nearest neighbor of v_i before v_{n+1} is added. v_{n+1} will become one of the *knn* of v_i if $\Delta_{i,\tau_i} > \Delta_{i,n+1}$. When this happens, v_{τ_i} is no longer one of the *knn* of v_i, and $e(i, \tau_i)$ should be broken if v_i is also not one of the *knn* of v_{τ_i}. This can be detected by checking if $\Delta_{\tau_i,i} > \Delta_{\tau_i,\iota_i}$ is true or not. Here, ι_i denotes the k-th nearest neighbor of v_{τ_i} after inserting v_{n+1}. We have

$$\begin{aligned} A = \{&e(i, n+1) : i \text{ is one of the } knn \text{ of } v_{n+1} \\ &\text{or } \Delta_{i,\tau_i} > \Delta_{i,n+1}\} \\ D = \{&e(i, \tau_i) : \Delta_{i,\tau_i} > \Delta_{i,n+1} \text{ and } \Delta_{\tau_i,i} > \Delta_{\tau_i,\iota_i}\}. \end{aligned} \tag{A.2}$$

A.1 Complexity The construction of A and D takes $O(n)$ time by checking these conditions for all the vertices. For *knn*-neighborhood, we need to find ι_i for all i. By examining all the neighbors of different vertices, we can find ι_i with time complexity $O(\sum_{i=1}^{n} deg(v_i) + |A|)$, which is just $O(|E| + |A|)$. $deg(v_i)$ denotes the degree of v_i. The complexity of this step can be bounded by $O(nq)$, where q is the maximum degree of the vertices in the graph after inserting v_{n+1}. Note that ι_i becomes the new τ_i for the $n+1$ vertices.

Appendix B Effect of Edge Deletion

Suppose we want to delete $e(a, b)$ from the graph. The lemma below is straightforward.

LEMMA B.1. *If $\pi_{ab} \ne a$, deletion of $e(a, b)$ does not affect any of the existing shortest paths and therefore no geodesic distance g_{ij} needs to be updated.*

For the remainder of this section we assume $\pi_{ab} = a$. This implies $\pi_{ba} = b$. The next lemma is an easy consequence of this assumption.

LEMMA B.2. *For any vertex v_i, $sp(i, b)$ passes through v_a iff $sp(i, b)$ contains $e(a, b)$ iff $\pi_{ib} = a$.*

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited

Let $R_{ab} \equiv \{i : \pi_{ib} = a\}$. Intuitively, R_{ab} contains vertices whose shortest paths to v_b include $e(a,b)$. We shall first construct R_{ab}, and then "propagate" from R_{ab} to get the geodesic distances that require update.

B.1 Construction Step Let $\mathcal{T}_{sp}(b)$ denote "the shortest path tree" of v_b, which is defined to consist of edges in the shortest paths with v_b as starting vertex. For any vertex v_t, $sp(t,b)$ consists of edges in $\mathcal{T}_{sp}(b)$ only. So the vertices in $sp(t,b)$, except v_t, are exactly the ancestors of v_t in the tree.

Lemma B.3. *R_{ab} is exactly the set of vertices in the subtree of $\mathcal{T}_{sp}(b)$ rooted at v_a.*

Proof.

$$\begin{aligned} & v_t \text{ is in the subtree of } \mathcal{T}_{sp}(b) \text{ rooted at } v_a \\ \Leftrightarrow\ & v_a \text{ is an ancestor of } v_t \text{ in } \mathcal{T}_{sp}(b) \\ \Leftrightarrow\ & sp(t,b) \text{ passes through } v_a \\ \Leftrightarrow\ & \pi_{tb} = a \qquad \text{(lemma B.2)} \\ \Leftrightarrow\ & t \in R_{ab} \end{aligned}$$

If v_t is a child of v_u in $\mathcal{T}_{sp}(b)$, v_u is the vertex in $sp(b,t)$ just before v_t. Thus, we have the lemma below.

Lemma B.4. *The set of children of v_u in $\mathcal{T}_{sp}(b) = \{v_t : v_t$ is a neighbor of v_u and $\pi_{bt} = u\}$.*

Consequently, we can examine all neighbors of v_u to find its children in $\mathcal{T}_{sp}(b)$. This leads to algorithm 1 that performs a tree traversal to construct R_{ab}.

B.1.1 Complexity At any time, the vertices in the queue Q are the examined vertices in the subtree. The while loop is executed $|R_{ab}|$ times. The inner for loop is executed a total of $\sum deg(v_t)$ times, where the summation is over all $v_t \in R_{ab}$. The sum can be bounded loosely by $q|R_{ab}|$. Therefore, a loose bound for algorithm 1 is $O(q|R_{ab}|)$.

```
R_ab := ∅; Q.enqueue(a);
while Q.notEmpty do
  t := Q.pop; R_ab = R_ab ∪ {t};
  for all v_u adjacent to v_t in G do
    if π_ub = t then
      Q.enqueue(u);
    end if
  end for
end while
```

Algorithm 1: Constructing R_{ab} by tree traversal.

B.2 Propagation Step We proceed to consider $F_{(a,b)} \equiv \{(i,j) : sp(i,j)$ contains $e(a,b)\}$. Note that (a,b) denotes an unordered pair, and $F_{(a,b)}$ is also a set of unordered pairs. $F_{(a,b)}$ contains vertex pairs such that the corresponding geodesic distances need to be recomputed when $e(a,b)$ is broken. $F_{(a,b)}$ is found by a search starting from v_b for each of the vertex in R_{ab}. R_{ab} and $F_{(a,b)}$ are related by the following two lemmas.

Lemma B.5. *If $(i,j) \in F_{(a,b)}$, either i or j is in R_{ab}.*

Proof. $(i,j) \in F_{(a,b)}$ means that $sp(i,j)$ contains $e(a,b)$. We can write $sp(i,j) = v_i \rightsquigarrow v_a \rightarrow v_b \rightsquigarrow v_j$ or $sp(i,j) = v_i \rightsquigarrow v_b \rightarrow v_a \rightsquigarrow v_j$, where $\rightsquigarrow$ denotes a path between the two vertices. Because the subpath of a shortest path is also a shortest path, either $sp(i,b)$ or $sp(j,b)$ passes through v_a. By lemma B.2, either $\pi_{ib} = a$ or $\pi_{jb} = a$. Hence either i or j is in R_{ab}.

Lemma B.6. *$F_{(a,b)} = \cup_{u \in R_{ab}} \{(u,t) : v_t$ in the subtree of $\mathcal{T}_{sp}(u)$ rooted at $v_b\}$.*

Proof. By lemma B.5, $(u,t) \in F_{(a,b)}$ implies either u or t is in R_{ab}. Without loss of generality, suppose $u \in R_{ab}$. So, $sp(u,t)$ can be written as $v_u \rightsquigarrow v_a \rightarrow v_b \rightsquigarrow v_t$. Thus v_t must be in the subtree of $\mathcal{T}_{sp}(u)$ rooted at v_b. On the other hand, for any vertex v_t in the subtree of $\mathcal{T}_{sp}(u)$ rooted at v_b, $sp(u,t)$ goes through v_b. Since $sp(u,b)$ goes through v_a (because $u \in R_{ab}$), $sp(u,t)$ must also go through v_a and hence use $e(a,b)$.

The above lemma seems to suggest that we need to construct different shortest path trees for different u in R_{ab}. This is not necessary because of the lemma below.

Lemma B.7. *Consider $u \in R_{ab}$. The subtree of $\mathcal{T}_{sp}(u)$ rooted at v_b is not empty, and let v_t be any vertex in this subtree. Let v_s be a child of v_t in the subtree, if any. We have the following:*

1. *v_t is in the subtree of $\mathcal{T}_{sp}(a)$ rooted at v_b.*
2. *v_s is a child of v_t in the subtree of $\mathcal{T}_{sp}(a)$ rooted at v_b*
3. *$\pi_{us} = \pi_{as} = t$*

Proof. The subtree of $\mathcal{T}_{sp}(u)$ rooted at v_b is not empty because v_b is in this subtree. For any v_t in this subtree, $sp(u,t)$ passes through v_b. Hence $sp(u,b)$ is a subpath of $sp(u,t)$. Because $u \in R_{ab}$, $sp(u,b)$ passes through v_a. So, we can write $sp(u,t)$ as $v_u \rightsquigarrow v_a \rightarrow v_b \rightsquigarrow v_t$. Thus $sp(a,t)$ contains v_b, implying that v_t is in the subtree of $\mathcal{T}_{sp}(a)$ rooted at v_b.

Now, if v_s is a child of v_t in the subtree of $\mathcal{T}_{sp}(u)$ rooted at v_b, $sp(u,s)$ can be written as $v_u \rightsquigarrow v_a \rightarrow$

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited

$v_b \rightsquigarrow v_t \to v_s$. So, $\pi_{us} = t$. Because any subpath of a shortest path is also a shortest path, $sp(a,s)$ is simply $v_a \to v_b \rightsquigarrow v_t \to v_s$, which implies v_s is also a child of v_t in $\mathcal{T}_{sp}(a)$ rooted at v_b, and $\pi_{as} = t$. Therefore, we have $\pi_{us} = \pi_{as} = t$.

Let F be the set of unordered pair (i,j) such that a new shortest path from v_i to v_j is needed when edges in D are removed. It is obvious that $F = \cup_{e(a,b)\in D} F_{(a,b)}$. F is constructed by merging different $F_{(a,b)}$, and $F_{(a,b)}$ can be obtained by algorithm 2. At each step, we traverse the subtree of $\mathcal{T}_{sp}(a)$ rooted at v_b, using the condition $\pi_{us} = \pi_{as}$ to check if v_s is in $\mathcal{T}_{sp}(u)$ rooted at v_b or not. The subtree of $\mathcal{T}_{sp}(a)$ is expanded "on-the-fly" by $\mathcal{T}'$.

```
F_(a,b) := ∅;
Initialize T', the expanded part of the subtree of
T_sp(a) rooted at v_b, to contain v_b only.
for all u ∈ R_ab do
  Q.enqueue(b)
  while Q.notEmpty do
    t := Q.pop;
    if π_at = π_ut then
      F_(a,b) = F_(a,b) ∪ {(u,t)};
      if v_t is a leaf node in T' then
        for all v_s adjacent to v_t do
          Insert v_s as a child of v_t in T' if π_as = t
        end for
      end if
      Insert all the children of v_t in T' to the queue
      Q;
    end if
  end while
end for
```

Algorithm 2: The algorithm to construct $F_{(a,b)}$.

B.2.1 Complexity If we ignore the time to construct $\mathcal{T}'$, the complexity of this step is proportional to the number of vertices examined. If the maximum degree of $\mathcal{T}'$ is q', this is bounded by $O(q'|F|)$. Note that $q' \leq q$. The time to expand $\mathcal{T}'$ is proportional to the number of vertices actually expanded plus the number of edges incident on those vertices. Thus, it is bounded by q times the size of the tree, and the size of the tree is at most of the same order as $|F_{(a,b)}|$. Usually, the time is much less, because different u in R_{ab} can reuse the same $\mathcal{T}'$. The time complexity to construct $F_{(a,b)}$ can be bounded by $O(q|F_{(a,b)}|)$ in the worst case. The overall time complexity to construct F, which is the union of $F_{(a,b)}$ for all $(a,b) \in D$, is $O(q|F|)$, assuming the number of duplicate pairs in $F_{(a,b)}$ for different (a,b) is $O(1)$. Empirically, there are at most several such duplicate pairs, while most of the time there is no duplicate pair at all.

Appendix C Updating the Geodesic Distances

Let $\mathcal{G}' = (V, E/D)$, the graph after deleting the edges in D. Let $\mathcal{A}$ be an undirected graph with the same vertices as $\mathcal{G}$ but with edges in F, i.e., $\mathcal{A} = (V,F)$. In other words, v_i and v_j are adjacent in $\mathcal{A}$ iff g_{ij} needs to be updated. Define $C_u = \{i : e(i,u)$ is an edge in $\mathcal{A}\}$. Our update strategy is to pick $v_u \in \mathcal{A}$ and then find the shortest paths from v_u to all vertices represented in C_u. This update effectively removes v_u from $\mathcal{A}$. We then pick another vertex $v_{u'}$ from $\mathcal{A}$, find the new shortest paths from $v_{u'}$, and so on, until there are no more edges in $\mathcal{A}$.

The new shortest paths are found by algorithm 3, which is based on the Dijkstra's algorithm with v_u as the source vertex. Recall the basic idea of Dijkstra's algorithm is to add vertex one by one to a "processed" set, in an ascending order of estimated shortest path distances. In our case, any vertex that is not in C_u is regarded as "processed", because its shortest path distance has already been computed and no modification is needed. The first "for" loop in algorithm 3 estimates the shortest path distances for vertices in C_u if the shortest paths are just "one edge away" from the "processed" vertices. In the while loop, the vertex with the smallest estimated shortest path distance is "processed", and we relax the estimated shortest path distances for the other "unprocessed" vertices accordingly.

C.1 Complexity The "for" loop takes at most $O(q|C_u|)$ time. In the "while" loop, there are $|C_u|$ ExtractMin operations, and the number of DecreaseKey operations depends on how many edges are there within the vertices in C_u. A upper bound for this is $q|C_u|$. By using Fibonacci's heap, ExtractMin can be done in $O(\log |C_u|)$ time while DecreaseKey can be done in $O(1)$ time, on average. Thus the complexity of algorithm 3 is $O(|C_u| \log |C_u| + q|C_u|)$. If binary heap is used instead, the complexity is $O(q|C_u| \log |C_u|)$.

C.2 Order of Update We have not yet discussed how to choose the vertex to be removed from $\mathcal{A}$ (in order to update its geodesic distances). Obviously, we should remove vertices in an order that minimizes the complexity of all the updates. Let f_i be the degree of the i-th vertex removed in $\mathcal{A}$. The overall time complexity of running the modified Dijkstra's algorithm for each of the removed vertices is $O(\sum_{i=1}^{n}(f_i \log f_i + q f_i))$. Because $\sum_{i=1}^{n} f_i$ is constant, we should delete the vertices

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited

for all $j \in C(u)$ **do**
  $H :=$ the set of indices of vertices that are adjacent to v_j in $\mathcal{G}'$ and not in $C(u)$;
  Insert $\delta(j) = \min_{k \in H} (g_{uk} + w_{kj})$ to a heap with index j. If $H = \emptyset$, $\delta(j) = \infty$.
end for
while Heap not empty **do**
  $k :=$ the index of the entry by "Extract Min" on the heap;
  $C(u) := C(u)/\{k\}; g_{uk} := \delta(k); g_{ku} := \delta(k)$;
  for all v_j that are adjacent to v_k in G' and $j \in C(u)$ **do**
    $dist := g_{uk} + w_{kj}$;
    if $dist < \delta(j)$ **then**
      "Decrease Key" from $\delta(j)$ to $dist$ for the entry with index j in the heap;
    end if
  end for
end while

Algorithm 3: Modified Dijkstra's algorithm for updating the geodesic distances.

in $\mathcal{A}$ in an order that minimizes $\sum_i f_i \log f_i$. However, finding the order to delete the vertices that minimizes the sum is hard. Since the sum is dominated by the largest f_i, we instead try to minimizes $\max_i f_i$. This can be done by a greedy algorithm that removes the vertex in $\mathcal{A}$ with the smallest degree. The correctness of this greedy approach can be seen from the following argument. Suppose the greedy algorithm is wrong. Then at some point the algorithm makes a mistake, i.e., it removes v_t instead of v_u, and the removal of v_t leads to an increase of $\max_i f_i$ from the smallest possible value. This can only happen when $deg(v_t) > deg(v_u)$. This is a contradiction, since the algorithm always removes the vertex with the smallest degree. Because the degree of each vertex is an integer, we can use an array of linked list to implement the greedy algorithm (algorithm 4).

Initialize an array of linked list $l[i]$ such that $l[i]$ is empty for $i = 1, \ldots, n$.
for all $v_u \in \mathcal{A}$ **do**
  $f :=$ degree of v_u in $\mathcal{A}$. Insert v_u to $l[f]$.
end for
$pos := 1$;
for $i := 1$ to n **do**
  If $l[pos]$ is empty, increment pos by one and until $l[pos]$ is not empty.
  Remove v_u, a vertex in the linked list stored in $l[pos]$, from the graph $\mathcal{A}$.
  Call the modified Dijkstra's algorithm for v_u and its neighbor (as C_u).
  for all v_j that is a neighbor of v_u **do**
    Find where v_j resides among the linked lists. This can be done by an indexing array.
    Move v_j from $l[f]$ to $l[f-1]$, or remove v_j from the linked lists if $f = 1$.
    If $f - 1 < pos$, set $pos := f - 1$.
  end for
end for

Algorithm 4: A greedy algorithm to determine a good order to remove the vertices in $\mathcal{A}$.

C.2.1 Complexity The first "for" loop takes $O(n)$ time. In the second "for" loop, pos is incremented at most $2n$ times, because it can at most move backwards n steps. The inner "for" loop is executed altogether $O(|F|)$ time. Therefore, the overall time complexity for algorithm 4 (excluding the time for executing the modified Dijkstra's algorithm) is $O(|F|)$.

Appendix D Shortening of Geodesic Distances

Recall A is the set of edges to be added to the graph. This is the same as the set of edges that are incident on v_{n+1} in $\mathcal{G}'$. The geodesic distances between v_{n+1} and other vertices are first found in $O(n|A|)$ time by the following equation:

$$g_{n+1,i} = g_{i,n+1} = \min_{\substack{j \text{ such that} \\ e(n+1,j) \in A}} \left(g_{ij} + w_{j,n+1} \right) \quad \forall i. \tag{D.1}$$

We proceed to consider how the addition of edges in A can decrease the other geodesic distances. Let $L = \{(i,j) : e(i, n+1)$ and $e(n+1, j)$ form a shorter path from v_i to v_j than $sp(i,j)\}$. Intuitively, L is the set of unordered pairs adjacent to v_{n+1} that new shortest paths running through v_{n+1} form. L can be constructed in $O(|A|^2)$ time.

We run algorithm 5 for different $(a,b) \in L$ in order to propagate the effect of the new shortest paths to other vertices. Suppose a better shortest path between v_i and v_j now emerges because the shortest path distance from v_a to v_b is reduced. We can find all such v_i and v_j in a manner similar to the construction of $F_{(a,b)}$. Without loss of generality, the new shortest path between v_i and v_j can be written as $v_i \rightsquigarrow v_a \rightarrow v_{n+1} \rightarrow v_b \rightsquigarrow v_j$. So, v_i is a vertex in the subtree of $\mathcal{T}_{sp}(n+1)$ rooted at v_a, and the first "while" loop in algorithm 5 locates all the vertices in the subtree, which are candidates for v_i. For any v_i, v_j must be in the "revised" subtree of $\mathcal{T}_{sp}(i)$ rooted at v_b. Here, "revised" means that the shortest path tree is the new tree that includes v_{n+1}. If v_j is in the "revised" subtree, v_j must be in the subtree of $\mathcal{T}_{sp}(n+1)$ rooted at v_b. Furthermore, if v_l is a child of v_j in the "revised" subtree, v_l must also be a child

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited

of v_j in the subtree of $\mathcal{T}_{sp}(n+1)$ rooted at v_b, and the condition $(g_{i,n+1} + g_{n+1,l}) < g_{il}$ must be true. The proof of these properties is similar to the proof for the relationship between $F_{(a,b)}$ and R_{ab} and hence is not repeated. These properties also explain the correctness of algorithm 5.

```
S := ∅; Q.enqueue(a);
while Q.notEmpty do
  t := Q.pop; S := S ∪ {t};
  for all v_u that are children of v_t in T_sp(n+1) do
    if g_{u,n+1} + w_{n+1,b} < g_{u,b} then
      Q.enqueue(u);
    end if
  end for
end while
for all u ∈ S do
  Q.enqueue(b);
  while Q.notEmpty do
    t := Q.pop; g_{ut} := g_{tu} := g_{u,n+1} + g_{n+1,t};
    for all v_s that are children of v_t in T_sp(n+1)
    do
      if g_{s,n+1} + w_{n+1,a} < g_{s,a} then
        Q.enqueue(s);
      end if
    end for
  end while
end for
```

Algorithm 5: Construction of shortest paths that are shortened because of v_{n+1}.

D.1 Complexity Let $H = \{(i,j)$: A better shortest path appears between v_i and v_j because of v_{n+1} $\}$. By an argument similar to the complexity of constructing F, we can see that the complexity of finding H and then revising the corresponding geodesic distances in algorithm 5 is $O(q|H| + |A|^2)$. The $O(|A|^2)$ time is due to the construction of L.

Appendix E Overall Complexity for Geodesic Distance Update

The neighborhood graph update takes $O(nq)$ time. The construction of R_{ab} takes $O(q|R_{ab}|)$ time, while the construction of F_{ab} takes $O(q|F_{ab}|)$ time. Since $|F_{ab}| \geq |R_{ab}|$, the last two steps take $O(q|F_{ab}|)$ time together. As a result, the time to construct F based on the removed and inserted edges is $O(q|F|)$. The time to run the Dijkstra's algorithm is difficult to estimate. Let μ be the number of vertices in $\mathcal{A}$ that have edges incident on them, and let $\nu \equiv \max_i f_i$ as defined in Appendix C. In the worst case, ν can be as large as μ, though this is highly unlikely. To get a glimpse of the typical values of ν, we can utilize concepts from random graph theory. It is easy to see that $\nu = \max_l\{\mathcal{A}$ has a l-regular sub-graph$\}$. Unfortunately, we have not been able to locate any result on the behavior of the largest l-regular sub-graph in random graphs. On the other hand, the properties of the largest l-complete sub-graph, i.e., a clique of size l, have been well studied for random graphs. The clique number (the size of the largest clique in a graph) of almost every graph is "close" to $O(\log \mu)$ [18]. We conjecture that, on average, ν is also of the order $O(\log \mu)$. This is in agreement with what we have observed empirically. Under this conjecture, the total time to run the Dijkstra's algorithm can be bounded by $O(\mu \log \mu \log\log \mu + q|F|)$. Finally, the time complexity of algorithm 5 is $O(q|H|+|A|^2)$. So, the overall time can be written as $O(q|F| + q|H| + \mu \log \mu \log \log \mu + |A|^2)$. In practice, the first two terms dominate, and we can write the complexity as $O(q(|F| + |H|))$.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited