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 actually 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 an 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

\(^1\)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
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 non-linear 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 $y_1, \ldots, y_n$ 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 $x_i \in \mathbb{R}^d$ be the co-ordinates corresponding to $y_i$. ISOMAP attempts to recover an isometric mapping from the coordinate space to the manifold. One may view $x_i$ as the (nonlinearly) reduced dimension representation of $y_i$. Define $X = (x_1, \ldots, x_n)$. Let $\Delta_{ij}$ be the distance between $y_i$ and $y_j$. ISOMAP also requires the user to specify the neighborhood. It can either be $\epsilon$-neighborhood, where $y_i$ and $y_j$ are neighbors if $\Delta_{ij}$ is less than a parameter $\epsilon$, or $k$nn-neighborhood, where $y_i$ and $y_j$ are neighbors if $y_i$ ($y_j$) is one of the $k$ nearest neighbors (knn) of $y_j$ ($y_i$). The value of $k$ is specified by the user.

The ISOMAP algorithm first constructs a weighted undirected neighborhood graph $G = (V, E)$ with the vertex $v_i \in V$ corresponding to $y_i$. An edge between $v_i$ and $v_j$, $e(i, j)$, exists if $y_i$ is a neighbor of $y_j$. The weight of $e(i, j)$, $w_{ij}$, is simply $\Delta_{ij}$. 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 $\pi_{ij}$, 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 $y_i$ and $y_j$, we shall call $g_{ij}$ “geodesic distance”. Note that $G = \{g_{ij}\}$ is a symmetric matrix. By assuming $\sum_i x_i = 0$, the target inner product matrix $B$ can be found by $B = HGH$, where $H = \{h_{ij}\}$, $h_{ij} = \delta_{ij} - 1/n$ and $\delta_{ij}$ is the delta function, i.e., $\delta_{ij} = 1$ if $i = j$ and 0 otherwise. We seek $X^T X$ to be as close to $B$ as possible by setting $X = (\sqrt{\lambda_1}v_1 \ldots \sqrt{\lambda_d}v_d)^T$, where $\lambda_1, \ldots, \lambda_d$ are the $d$ largest eigenvalues of $B$, with corresponding eigenvectors $v_1, \ldots, v_d$. Note that computing $GHG$ is effectively a centering operation on $G$, and this can be computed in $O(n^2)$ time.

3 Incremental Version of ISOMAP

Suppose we have the co-ordinates $x_i$ of $y_i$ for $1 \leq i \leq n$. The new sample $y_{n+1}$ arrives and the goal of incremental ISOMAP is to update the co-ordinates $x_i$ 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 $x_1, \ldots, x_n$ are then updated because of the changes in $g_{ij}$. Finally, $x_{n+1}$, 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 $y_{n+1}$ introduces a new vertex $v_{n+1}$ in the graph $G$. 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.
While the proposed algorithm is applicable for both knn and $\epsilon$ 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 $G_{\text{new}}$. One may view this as an incremental eigenvalue problem, as the co-ordinates $x_i$ 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 $G_{\text{new}}$ denote the matrix of updated geodesic distances. Given $B = HGH$ and $X$ such that $B \approx X^T X$, our goal is to find the new $X_{\text{new}}$ such that $X_{\text{new}}^T X_{\text{new}} \approx B_{\text{new}}$, where $B_{\text{new}} = HG_{\text{new}}H$. Our first approach is based on gradient descent. The eigen decomposition in batch ISOMAP is equivalent to finding $X$ that minimizes

$$J(B, X) = \text{tr}((B - X^T X)(B - X^T X)^T)/n^2,$$

which is the average of the square of the entries in $B - XX^T$. Its gradient is

$$\nabla_X J(B, X) = (-iX B + iXX^T X)/n^2,$$

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

Another approach to update $X$ is to find the eigenvalues and eigenvectors of $B_{\text{new}}$ by an iterative approach. We first recover (approximately) the eigenvectors of $B$ from $X$ by normalizing the $i$-th column of $X^T$ to norm one to obtain the $i$-th eigenvector $v_i$, and form $V = (v_1, \ldots, v_d)$ as a reasonable initial guess of the eigenvectors of $B_{\text{new}}$. Subspace iteration together with Rayleigh-Ritz acceleration [12] is used to refine $V$ as eigenvectors of $B_{\text{new}}$.

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

2. Form $Z = V^T B_{\text{new}} V$ and perform eigen-decomposition of the $d$ by $d$ matrix $Z$. Let $\lambda_i$ and $u_i$ be the $i$-th eigenvalue and the corresponding eigenvector.

3. $V_{\text{new}} = V[u_1 \ldots u_d]$ is the improved set of eigenvectors of $B_{\text{new}}$.

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

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

$$||x_{n+1}||^2 = \frac{1}{n} \sum_{i=1}^n (\gamma_i - \sum_{i=1}^n ||x_i||^2)$$

and $x_{n+1}^T x_i = -\frac{1}{2} (\gamma_i - ||x_{n+1}||^2 - ||x_i||^2)$  $\forall i$

By replacing $\gamma_i$ with the actual geodesic distance $g_i, n+1$, we obtain our target inner product between $x_{n+1}$ and $x_i, f_i$, in a manner similar to equation (3.3). $x_{n+1}$ can be found by solving (in least-square sense) the equation $X^T x_{n+1} = f$, where $f = (f_1, \ldots, f_n)^T$. Alternatively, we can initialize $x_{n+1}$ 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 $x_{n+1}$, we normalize them so that the center of all the $x_i$ 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 $O(n^2 \log n + n^2 q)$. 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
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.

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. 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.

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.

---

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

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.
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.

![Approximation error](image1)

(a) Approximation error

![Last snapshot](image2)

(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 $y_i$ 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 $x_i$, can be estimated from the data by examining the residue of $B - X^T X$ 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-
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


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 \(\epsilon\)-neighborhood, there is an edge between \(v_i\) and \(v_{n+1}\) if \(\Delta_{i,n+1} \leq \epsilon\). Also, \(\Delta_{ij}\) is not affected by \(v_{n+1}\) for \(1 \leq i \leq n\) and \(1 \leq j \leq n\). So no edge is deleted. Therefore,

\[
\begin{align*}
A &= \{ (i,i+1) : i \in 1,..., n \text{ and } \Delta_{i,n+1} \leq \epsilon \} \\
D &= \emptyset.
\end{align*}
\]

The case for \(knn\)-neighborhood is more complicated. \(e(i,i+1)\) 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 \(\tau_i\) 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, \(v_i\) is no longer one of the \(knn\) of \(v_{n+1}\) and \((i,\tau_i)\) should be broken if \(v_i\) is also not one of the \(knn\) of \(v_{n+1}\). This can be detected by checking if \(\Delta_{\tau_i,i} > \Delta_{\tau_i,n+1}\) is true or not. Here, \(\tau_i\) denotes the \(k\)-th nearest neighbor of \(v_i\) after inserting \(v_{n+1}\). We have

\[
\begin{align*}
A &= \{ (i,i+1) : i \text{ is one of the } knn \text{ of } v_{n+1} \text{ or } \Delta_{i,\tau_i} > \Delta_{i,n+1} \} \\
D &= \{ (i,\tau_i) : \Delta_{i,\tau_i} > \Delta_{i,n+1} \text{ and } \Delta_{\tau_i,i} > \Delta_{\tau_i,n+1} \}.
\end{align*}
\]

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 \(\tau_i\) for all \(i\). By examining all the neighbors of different vertices, we can find \(\tau_i\) with time complexity \(O(\sum_{i=1}^{n} \deg(v_i) + |A|)\), which is just \(O(|E| + |A|)\). \(\deg(n_i)\) 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 \(\tau_i\) becomes the new \(\tau_i\) 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 \(\pi_{ab} \neq a\), 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 \(\pi_{ib} = a\).
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 $T_{sp}(b)$ 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 $v$, $sp(t, b)$ consists of edges in $T_{sp}(b)$ 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 $T_{sp}(b)$ rooted at $v_a$.

**Proof.**
\[ v_t \text{ is in the subtree of } T_{sp}(b) \text{ rooted at } v_a \iff v_a \text{ is an ancestor of } v_t \text{ in } T_{sp}(b) \iff sp(t, b) \text{ passes through } v_a \iff \pi_{ib} = a \quad \text{(lemma B.2)} \iff t \in R_{ab} \]

If $v_t$ is a child of $v_a$ in $T_{sp}(b)$, $v_a$ 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 $v_a$ in $T_{sp}(b) = \{v_t : v_t \text{ is a neighbor of } v_a \text{ and } \pi_{it} = u\}.$

Consequently, we can examine all neighbors of $v_a$ to find its children in $T_{sp}(b)$. 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 $O(q|R_{ab}|)$.

```plaintext
R_{ab} := \emptyset; Q.enqueue(a);
while QnotEmpty do
  t := Q.pop; R_{ab} = R_{ab} \cup \{t\};
  for all v_t adjacent to v_t in $G$ do
    if $\pi_{ib} = t$ 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 $F_{(a, b)} \equiv \{(i, j) : sp(i, j) \text{ contains } e(a, b)\}$. 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 $sp(i, j) = v_i \sim v_a \sim v_b \sim v_j$ or $sp(i, j) = v_i \sim v_b \sim v_a \sim v_j$, where $\sim$ 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.** $F_{(a, b)} = \cup_{u \in R_{ab}} \{(u, t) : v_t \text{ in the subtree of } T_{sp}(u) \text{ rooted at } v_b\}$.

**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 $v_u \sim v_a \sim v_b \sim v_t$. Thus $v_t$ must be in the subtree of $T_{sp}(u)$ rooted at $v_b$. On the other hand, for any vertex $v_t$ in the subtree of $T_{sp}(u)$ rooted at $v_b$, $sp(u, t)$ goes through $v_a$. 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 $T_{sp}(u)$ rooted at $v_b$ is not empty, and let $v_s$ 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 $T_{sp}(a)$ rooted at $v_b$.
2. $v_s$ is a child of $v_t$ in the subtree of $T_{sp}(a)$ rooted at $v_b$.
3. $\pi_{as} = \pi_{at} = t$

**Proof.** The subtree of $T_{sp}(u)$ 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 $v_u \sim v_a \sim v_b \sim v_t$. Thus $sp(u, t)$ contains $v_a$, implying that $v_t$ is in the subtree of $T_{sp}(a)$ rooted at $v_b$.

Now, if $v_s$ is a child of $v_t$ in the subtree of $T_{sp}(u)$ rooted at $v_b$, $sp(u, s)$ can be written as $v_u \sim v_a \sim v_s \sim v_b \sim v_t$. For every $v_s \in R_{ab}$, $sp(u, s)$ contains $v_a$, implying that $v_t$ is in the subtree of $T_{sp}(a)$ rooted at $v_b$.

Thus, we can construct a rooted shortest path tree for each vertex $u \in R_{ab}$.
degree of to the number of vertices examined. If the maximum struct \( T \) is simply \( v_a \rightarrow v_b \rightarrow v_t \rightarrow v_s \), which implies \( v_s \) is also a child of \( v_t \) in \( T_{sp}(a) \) rooted at \( v_t \), and \( \pi_{as} = t \). Therefore, we have \( \pi_{as} = \pi_{as} = t \).

Let \( F \) be the set of unordered pair \((i,j)\) such that a new shortest path from \( v_t \) to \( v_j \) is needed when edges in \( D \) are removed. It is obvious that \( F = \cup_{c(a,b) \in D} F_{(a,b)} \). \( F \) 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 \( T_{sp}(a) \) rooted at \( v_b \), using the condition \( \pi_{as} = \pi_{as} \) to check if \( v_s \) is in \( T_{sp}(u) \) rooted at \( v_b \) or not. The subtree of \( T_{sp}(a) \) is expanded “on-the-fly” by \( T' \).

```
F_{(a,b)} := \emptyset;
Initialize T’, the expanded part of the subtree of T_{sp}(a) rooted at v_b, to contain v_b only.
for all u \in R_{ab} do
    Q_.enque(u)
while Q_.notEmpty do
    t := Q_.pop;
    if \( \pi_{at} = \pi_{ut} \) then
        F_{(a,b)} = F_{(a,b)} \cup \{(u,t)\};
        if \( v_t \) is a leaf node in T’ then
            for all v_t adjacent to v_t do
                Insert \( v_s \) as a child of \( v_t \) in T’ if \( \pi_{as} = t \)
            end for
        end if
        Insert all the children of \( v_t \) in T’ to the queue Q;
    end if
end while
```

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

B.2.1 Complexity If we ignore the time to construct \( T' \), the complexity of this step is proportional to the number of vertices examined. If the maximum degree of \( T' \) is \( q' \), this is bounded by \( O(q'|F|) \). Note that \( q' \leq q \). The time to expand \( T' \) 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 \( T' \). 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 \( G' = (V,E'/D) \), the graph after deleting the edges in \( D \). Let \( \mathcal{A} \) be an undirected graph with the same vertices as \( G \) 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 \( v_t \) 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.
for all \( j \in C(u) \) do
\[
H := \text{the set of indices of vertices that are adjacent to } v_j \text{ in } G' \text{ and not in } C(u);
\]
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
\[
k := \text{the index of the entry by “Extract Min” on the heap};
\]
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
\[
\text{“Decrease Key” from } \delta(j) \text{ to } dist \text{ for the entry with index } j \text{ in the heap};
\]
end if
end for
end while

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

We proceed to consider how the addition of edges in \( A \) can decrease the other geodesic distances. Let \( L = \{(i, j) : e(i, n + 1) \text{ and } e(n + 1, j) \text{ form a shorter path from } v_i \text{ to } v_j\} \). 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 \( v_i \rightarrow v_a \rightarrow v_{n+1} \rightarrow v_b \rightarrow v_j \). So, \( v_i \) is a vertex in the subtree of \( T_{sp}(n + 1) \) 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 \( T_{sp}(i) \) rooted at \( v_a \). 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 \( T_{sp}(n + 1) \) rooted at \( v_b \). Furthermore, if \( v_j \) is a child of \( v_i \) in the “revised” subtree, \( v_j \) must also be a child.
of \( v_j \) in the subtree of \( T_{sp}(n+1) \) rooted at \( v_i \), and the condition \( (g_{i,n+1} + g_{n+1,i}) < g_{i,i} \) 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.

\[
S := \emptyset; Q.\text{enqueue}(a); \\
\text{while } Q.\text{notEmpty} \text{ do} \\
\quad t := Q.\text{pop}; S := S \cup \{t\}; \\
\quad \text{for all } v_u \text{ that are children of } v_t \text{ in } T_{sp}(n+1) \text{ do} \\
\quad \quad \text{if } g_{u,n+1} + w_{n+1,b} < g_{u,b} \text{ then} \\
\quad \quad \quad Q.\text{enqueue}(u); \\
\quad \quad \text{end if} \\
\quad \text{end for} \\
\text{end while} \\
\text{for all } u \in S \text{ do} \\
\quad Q.\text{enqueue}(u); \\
\text{while } Q.\text{notEmpty} \text{ do} \\
\quad t := Q.\text{pop}; g_{ut} := g_{ut} := g_{u,n+1} + g_{n+1,i}; \\
\quad \text{for all } v_s \text{ that are children of } v_t \text{ in } T_{sp}(n+1) \text{ do} \\
\quad \quad \text{if } g_{s,n+1} + w_{n+1,a} < g_{s,a} \text{ then} \\
\quad \quad \quad Q.\text{enqueue}(s); \\
\quad \quad \text{end if} \\
\quad \text{end for} \\
\text{end while} \\
\text{end for}
\]

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

### D.1 Complexity

Let \( H = \{(i, j) : \text{A better shortest path appears between } v_i \text{ and } v_j \text{ because of } v_{n+1}\} \). 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 \( \mu \) be the number of vertices in \( A \) that have edges incident on them, and let \( \nu \equiv \max_i f_i \) as defined in Appendix C. In the worst case, \( \nu \) can be as large as \( \mu \), though this is highly unlikely.

To get a glimpse of the typical values of \( \nu \), we can utilize concepts from random graph theory. It is easy to see that \( \nu = \max_i \{|A| \text{ has a } l\text{-regular sub-graph}\} \). Unfortunately, we have not been able to locate any result on the behavior of the largest \( l\text{-complete sub-graph} \), 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, \( \nu \) 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 \( O(q(|F| + |H|)) \).