Path related problems
Constrained shortest path and the $k-$shortest paths
Application to ISOMAP

Mohamed Yahya Soali
Gabriel Govignon

18 May 2020

Contents

1 Introduction

The shortest path problem is one of the fundamental problems in computer science and graph theory. It consists of finding a path connecting a given source-destination pair in a network at minimum cost. The problem has many applications where the aim is to send some resources between two specific points as quickly, cheaply or reliably as possible. In the domain of data science, some of these applications are related to the reduction of dimensionality in datasets embedded in high-dimensional spaces.

This programming project deals with the conception and implementation of path-related algorithms. In the first part, we address the well-known problem of finding the shortest path in a given oriented or non-oriented weighted graph. In the second one, we add a resource count constraint. In the third part, we describe algorithms to find not only the very best path, but also the k-shortest paths. As an application of these tools, we will explore at the end the use of shortest paths in the dimensionality reduction of data with ISOMAP.

- For this project, we chose to code in Python for it's highly flexibility and plentiful useful libraries, making coding easier.

2 Graph model

In this section, we remind some graph theory related concepts and definitions and discuss the implementation methods.

- A graph is a pair $G = (V, E)$ where : $\begin{cases} V \text{ is a finite set of vertices or nodes.} \\ E \subset \{(u,v)/u, v \in V\} \text{ is a set of edges. (no loop)} \end{cases}$
- For some remarks about complexity, it is useful to introduce : $|V| = n$ and $|E| = m$.
- An edge is an arrow connecting two vertices, it indicates a single direction.
- A graph is said to be directed if : $(\forall (u,v) \in E),\ (v,u) \in E$.
- We distinguish (in a directed graph) two types of neighbors for each node v : In neighbors : $N^-(v) = \{u \in V/(u,v) \in E\}$. Out neighbors : $N^+(v) = \{u \in V/(v,u) \in E\}$ These sets are all the same for an undirected graph for each node.
- A graph is said to be weighted if there exists a cost function $w : E \longrightarrow \mathbb{R}$ that puts a weight for each edge.
- A graph is said to be connected if each two vertices are connected by at least one path.
- A path in a undirected weighted graph $G = (V, E)$ is a sequence $P\ :\ v_0, e_1, v_1, ..., v_{k-1}, e_k, v_k$ such that $v_0, ..., v_k \in V$, $e_1, ..., e_k \in E$, and $e_i = (v_{i-1}, v_i)\ \forall i \in [1..k]$

  - Moreover, P is a path connecting $s \in V$ and $t \in V$ if : $\begin{cases} s = v_0 \\ t = v_k \end{cases}$

  We say also that the P's length is : $w(P) = \sum_{i=1}^{k} w(e_i)$ (the sum of P's edges weights).

  - We say of a path connecting s to t that it is the shortest path if its length is not longer than the length of any other path connecting s and t.We denote then by $d(s,t)$ the length of a shortest path connecting s and t.
- For our implementation, we chose to use adjacency matrix to store the neighbours of every vertices (its coefficients are the weights in case of weighted graphs).

3 Shortest path problem

Our aim, in this section, is to compute the shortest path between a pair of vertices in an (un)directed connected weighted graph.

When we do not specify whether it is undirected... or directed..., it means that the following algorithms do not depend on it.

Let $G = (V, E)$ a connected weighted graph and $s, t \in V$ a source and target node successively.

3.1 Dijkstra's algorithm

3.1.1 Algorithm description

We assume here that the edge weights are non-negative, that is, $w : E \longrightarrow \mathbb{R}^+$. Computing a shortest path from some node s to another t is not significantly easier than computing a shortest path from s to all other nodes $u \in V$. Hence when talking about Dijkstra's algorithm, usually the single-source shortest path problem is referred to (the shortest path tree from s).

The two key ideas in Dijkstra's algorithm [1] are the following :

An exchange property : We first observe that when there is a shortest path P from s to t, then this immediately gives shortest paths from s to all nodes on P. This reminds us of dynamic programming: we may start calculating shortest paths incrementally.

Greediness : Assume that we already have a shortest path tree $T = (V_T, E_T)$ for a subset V_T of the vertices. For a vertex $u \in V \backslash V_T$ outside this tree, let us call $d_t(s, u)$ the tentative distance of u from s i.e. the minimal length of a path from s to u that is fully in T apart from the last edge :

$$\boxed{d_t(s,u) = \begin{cases} \min\{d(s,v) + w(v,u) / v \in V_T \cap N^-(u)\} \text{ if } N^-(u) \neq \emptyset \\ +\infty \text{ otherwise} \end{cases}}$$

if u is such that it has the minimal tentative distance among all vertices outside T, then its tentative distance is also its true distance and the path giving rise to the tentative distance is in fact a shortest path.

- We start with a tree consisting solely of s and then repeatedly add vertex with smallest tentative distance to T.

- Dijkstra's algorithm maintains a partition of V into settled (for which we did found the true distance), queued (already reached by a path from s but not sure if it is the shortest one) and unreached nodes as it is shown in 1.

We give a general pseudocode for Dijkstra's algorithm in 1.

3.1.2 Complexity

We have used an array of length n to store the predecessors of each node in the shortest path tree rooted at s.

For finding the vertex u with minimal tentative distance, we can either **use a list and sort its element with repect to their tentative distance, which will lead to a complexity of** $O(mnlog(n))$. The second choice is **to do so with a priority queue having hence** $(mlog(n))$.

In both approaches, we remind that the tentative distances are updated for all neighbors of the selected node in each step of the algorithm.

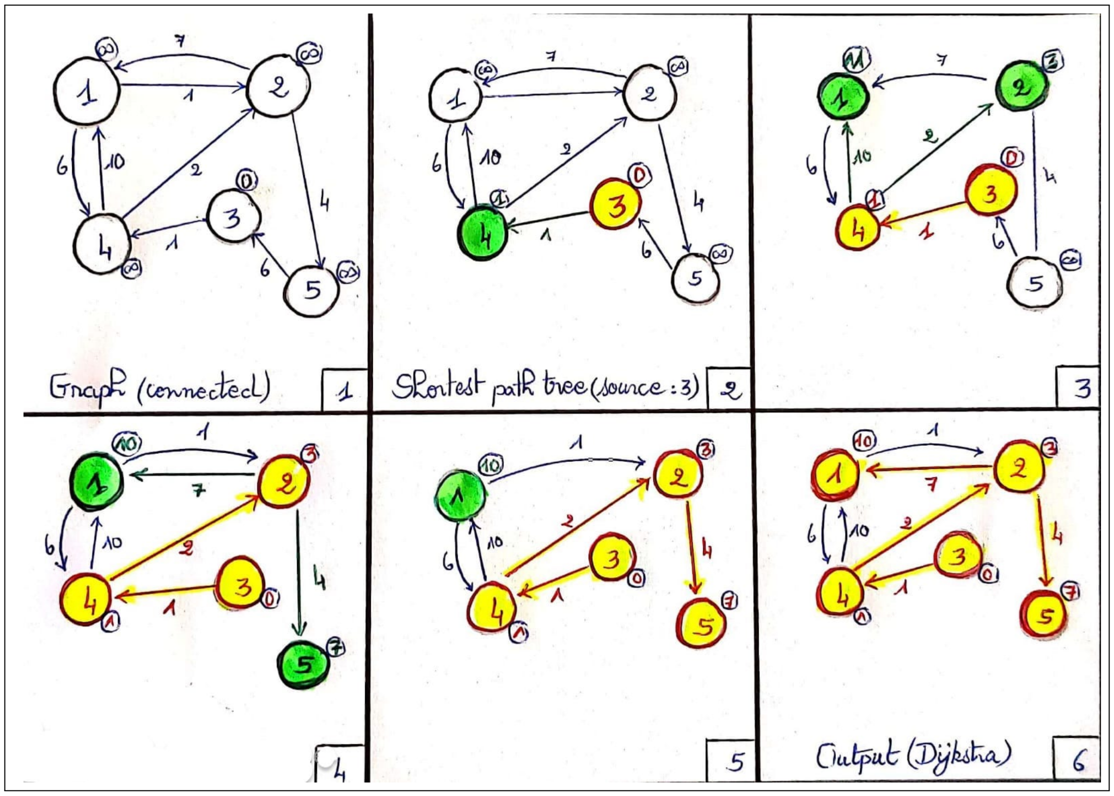


Figure 1: Dijkstra's algorithm steps on a graph with "3" as a source : The shortest path tree rooted at 3 and the settled nodes are colored in red, the queued nodes in green (with their corresponding edges), the unreached nodes in black and the tentative distances are inserted in small circles next to each node.

3.2 Bellman-Ford algorithm

3.2.1 Description

In this sub-section we do not assume that all edge weights are non-negative. We do, however, assume that there are no negative cycles. A negative cycle is a cycle in the graph such that the sum of the weights of its edges is less than zero. If we authorize negative cycles to exist, there will be obviously no shortest path.

We solve the shortest path problem in this case by using **dynamic programming**. Let's denote $d_i(s,v)$ the minimum length of a path from s to v that consists of at most i edges. We have the dynamic programming equations :

$$\begin{cases} d_0(s,s) = 0 \\ d_0(s,v) = +\infty \ (\forall v \neq s) \\ d_i(s,v) = \{d_{i-1}(s,v), d_{i-1}(s,u) + w(u,v) | u \in N^-(v)\} \ (\forall v \in V)(\forall i \in [\![1,..,n-1]\!]) \end{cases}$$

Since there are no paths with more than $n-1$ edges, we have $d_{n-1}(s,v) = d(s,v)$.

Remark that since $d_i(s,v)$ depends only on distances d_{i-1}, we only need to store $d_{i-1}(s,.)$ and $d_i(s,.)$. This allow us to have a $O(n)$ space complexity. We add also that an intelligent implementation will see that a simple array of distances d is enough for the algorithm as the Bellman-Ford approach shows in the pseudocode 2.

3.2.2 Complexity

This algorithm performs two nested loops; one on $i \in [\![1, .., n-1]\!]$ and the other on the neighbors of each vertex. The time complexity is $O(mn)$. The spatial complexity is of $O(n)$ as explained before.

3.3 A parallel algorithm

Theoretically, **Dijkstra's algorithm is the most efficient sequential algorithm on graphs with non-negative edge weights**. Sequential means here that vertices are picked from the queue successively.

We discuss below a parallel version of the dijkstra's algorithm [2] which is better fast and work-efficient than the original one in some cases.

3.3.1 Description

The queue in Dijkstra's algorithm may contain more than one node v with $d_t(s, v) = d(s, v)$, all such nodes could be removed simultaneously if identified. There are several sufficient criteria that help distinguish such nodes namely:

The OUT-version : we compute a treshold defined via the weights of outgoing edges : $L = min\{d_t(s, u) + w(u, z)/(z \in N^+(u))\ u \text{ is queued}\}$, and remove all nodes v from the queue which satisfy $d_t(s, v) \leqslant L$.

Others like the IN-version and INOUT exists as well (see [2]).

3.3.2 Efficiency

The number of steps (phases) in the parallel algorithm of Dijkstra can decrease considerably in comparison with the original algorithm, especially when running on random graphs which edges are included with a probability of $\frac{C}{n}$, (C a constant) with edge weights randomly chosen with uniform distribution in $[0, 1]$. **The algorithm performs a $O(\sqrt{n})$ time complexity better than the sequential one in this case** (see 2).

3.4 All pairs shortest paths

We finally describe an algorithm to compute all the shortest paths lengths between vertices. This one would be useful in the implementation of ISOMAP dimensionality reduction algorithm.

We can run Dijkstra's algorithm or Bellman-Ford over all vertices (n times), **this will end up with a $O(mnlog(n))$ time complexity or $O(n^2m)$ successively**. There is a simpler much better

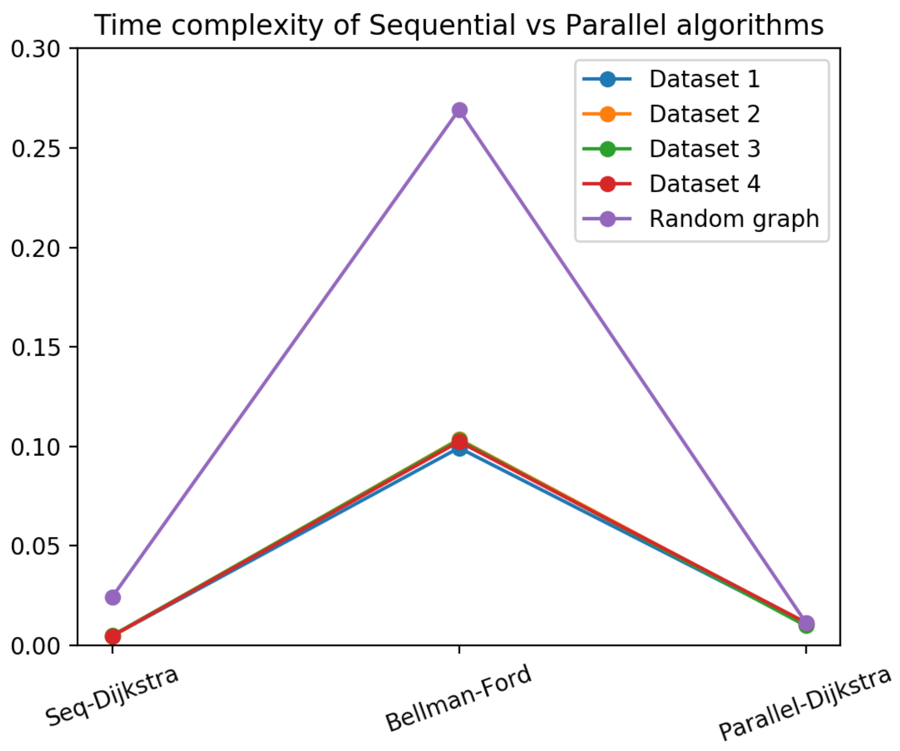


Figure 2: Time complexity of shortest path algorithms runned on the dataset [3] provided in and on a random graph.

algorithm than the Bellman-Ford in case of negative weights : **The Floyd-Warshall algorithm having a complexity of $O(n^3)$** which pseudocode is described in 3.

4 Constrained shortest paths

The shortest paths problem is the problem of computing a shortest path connecting s and t with some additional resources constraints : going through an edge has a given cost in every K sorts of resources and the sum of a resource costs through a path should be between the resource tresholds. In order to solve this problem, we formulate it as an integer linear programming problem (ILP).
We denote :

K : the number of resources.

$m_1, \dots, m_K$: the minimum amount of each resource spent.

$M_2, \dots, M_K$: the maximum amount of each resource spent.

$w(i,j)$: the weight of the edge (i,j).

$c_k(i,j)$: the cost in terms of resource k to go through edge (i,j).

$d(i,j)$: the distance between nodes i and j

We define **the binary variable** $x_{i,j} \in \{0,1\}$ **which are equals to** 1 **when the constrained shortest path between** s **and** t **does contain the edge** (i,j). (We implicitly consider $V = \{1,2,...,n\}$). The problem can be formulated as follows:

$$\begin{cases} \min\limits_{x_{i,j}} \left[\sum\limits_{i=1}^{n} \sum\limits_{j=1}^{n} w(i,j) x_{i,j} \right] \\ s.to \\ \sum\limits_{j=1}^{n} x_{s,j} - \sum\limits_{j=1}^{n} x_{j,s} = 1 \\ \sum\limits_{j=1}^{n} x_{t,j} - \sum\limits_{j=1}^{n} x_{j,t} = -1 \\ \sum\limits_{j=1}^{n} x_{i,j} - \sum\limits_{j=1}^{n} x_{j,i} = 0 \ (\forall i \in V \backslash \{s,t\}) \\ m_k \leq \sum\limits_{i,j} c_k(i,j) x_{i,j} \leq M_k \ (\forall k \in [\![1,..,K]\!]) \end{cases}$$

We remind the very attentive reader that the previous system with the last resources constraints omitted is equivalent to the shortest path problem.

Several well-known algorithms are already available to solve this ILP-problem. We cite here the simplex method based on the integrity constraints relaxation ($x_{i,j} \in [0,1]$) and the approximate solutions that goes with it.

5 k-shortest paths

Given a $k \in \mathbb{N}$, the k-shortest paths problem consists in enumerating the first k-shortest paths in the graph.

Based on Dijkstra's algorithm idea, we describe below an algorithm finding the k-shortest paths in a graph.

- We start with the path "$P_s : s$". At each iteration, we pick the shortest path P_u in the queue, increment the number of paths of the corresponding last node u of P_u and add new paths generated from P_u and another edge connecting u to its neighbors. We stop the algorithm once the queue is empty or the number of paths ending with t already found exceeds k.

We give the pseudo code in 4 for more details.

6 Dimensionality reduction with ISOMAP

The dimensionality reduction is one of the important data preprocessing technique for large scale classification data tasks. It consists in finding meaningful low-dimensional structures hidden in high dimensional observations, by reducing the number of random variables under consideration and obtaining a set of principal variables. This help reducing research costs and facilitate data manipulation.

Several classical techniques for **linear dimensionality reduction** exist namely PCA (principal component analysis) and MDS (multidimensional scaling) [4]. The algorithms are simple to implement, compute efficiently, and are guaranteed to discover the true structure of data lying on a linear subspace of the high dimensional input space. They find a low-dimensional representation of the data that retains as much information as possible by keeping important principle component. This information is for example contained in the co-variance matrix or in distance matrix of data. These algorithms do well on linear data structure. **However, many data sets contain essential nonlinear structures that are invisible to both PCA and MDS**.

- Here we describe an approach to solve nonlinear dimensionality reduction problems [5], capable of finding out the nonlinear degrees of freedom that underlies complex natural observations. This one provides a simple method for estimating the intrinsic geometry of a data manifold and representing it in a low dimensional euclidean space.

6.1 Beyond euclidean geometry

As we mentioned before, ISOMAP also deals with data embedded in a non-linear geometry space (non-euclidean distance). We call such a geometry a **"geodesic geometry"** [6] to distinguish it from the euclidean one. The distance hence between points in the data manifold can not be computed with the euclidean distance but with respect to the geometry of the space.

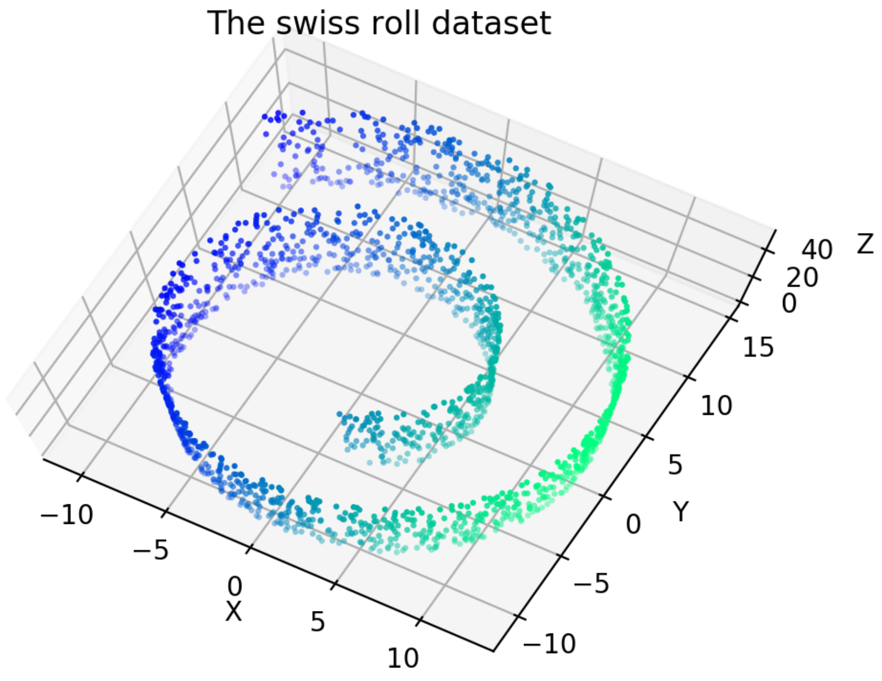


Figure 3: The Swiss roll dataset for the first 1000 points of the cloud. (Geodesic geometry).

The true distance between points is called the geodesic distance. In 3 dimensions, we talk about orthodromic distance (well-known in aeronautics). All the task will be to estimate this geodesic distance in nonlinear data manifolds. Once done, we could run the MDS algorithm on the geodesic matrix of distances to get a low-dimensional representation of data.

An example of a geodesic geometry different than the euclidean geometry is described by the 3-dimensional data manifold "swiss roll"[7].The euclidean distance between two points (in blue) seems to be shorter than the real geodesic one (in red); this induces serious errors (see Figure 4) .

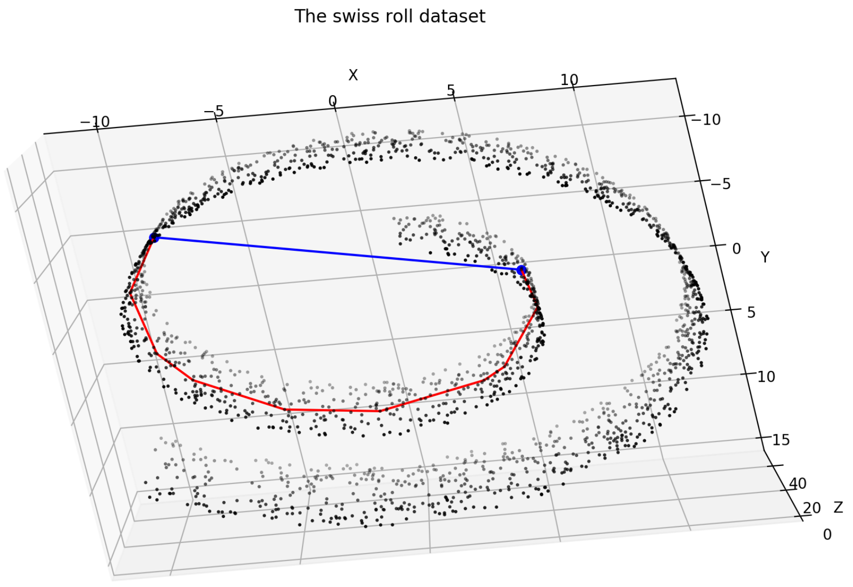


Figure 4: The distance between two points 1 and 101 in the cloud : euclidean in blue vs the geodesic one in red.

6.2 Description of ISOMAP algorithm

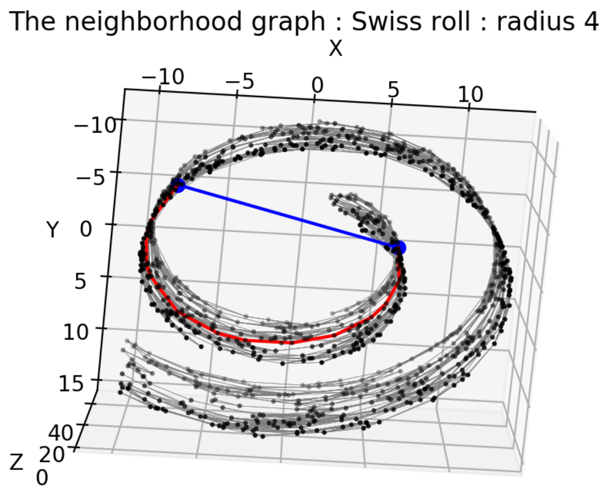
Suppose we have, as an input, a data manifold of N n-dimensional points and a d the desired dimension of the output ($d < n$).

The complete isometric feature mapping or ISOMAP algorithm has three main steps:

1 - Construction of the neighbourhood graph : Based on euclidean distances between pairs of points, we connect points which are very close to each other with edges having the distance between those points as weights. There exists two methods to link closest points :

- Connect each point to all points within some fixed radius ϵ.
- Connect each point to its K-nearest neighbours.

The art of this step will be to determine ϵ or K that better links the points: not too small so that **we obtain a connected graph** and not too large so that **the geodesic geometry is still respected** (see Figure 5).



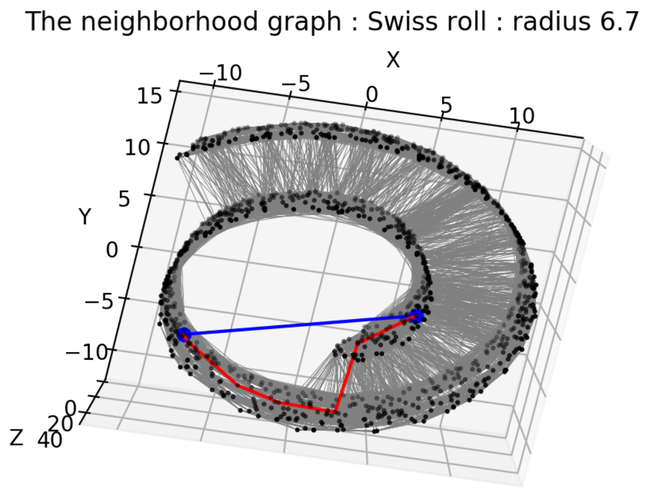


Figure 5: The neighborhood graph for the first 1000 points of the Swiss roll cloud.

2 - Computation of shortest paths : In this step, we calculate the geodesic distances matrix between points of the manifold. For neighbouring points, this distance will be the edge weight. For faraway points, it is the cost of the shortest path linking them in the neighbourhood graph. We use to achieve this a Floyd-Warshall algorithm 3 .

3 - Dimensionality reduction : Construction of a d-dimensional embedding by using an MDS on the geodesic distances matrix.

The main points of ISOMAP are summed up in the pseudo code below (ϵ method) 5:

- As an output, we obtain a representation of data points in a euclidean low-dimensional space that preserve the best it can the initial information about the data manifold and converge to the true structure (see Figure 6) .

This problem is of major and central importance in many fields; in vision, speech analysis or other subjects related to the detection of data non-linearities. ISOMAP provides an efficient way to visualize and analyze data while retaining their intrinsic properties. You can refer to [5] for more details.

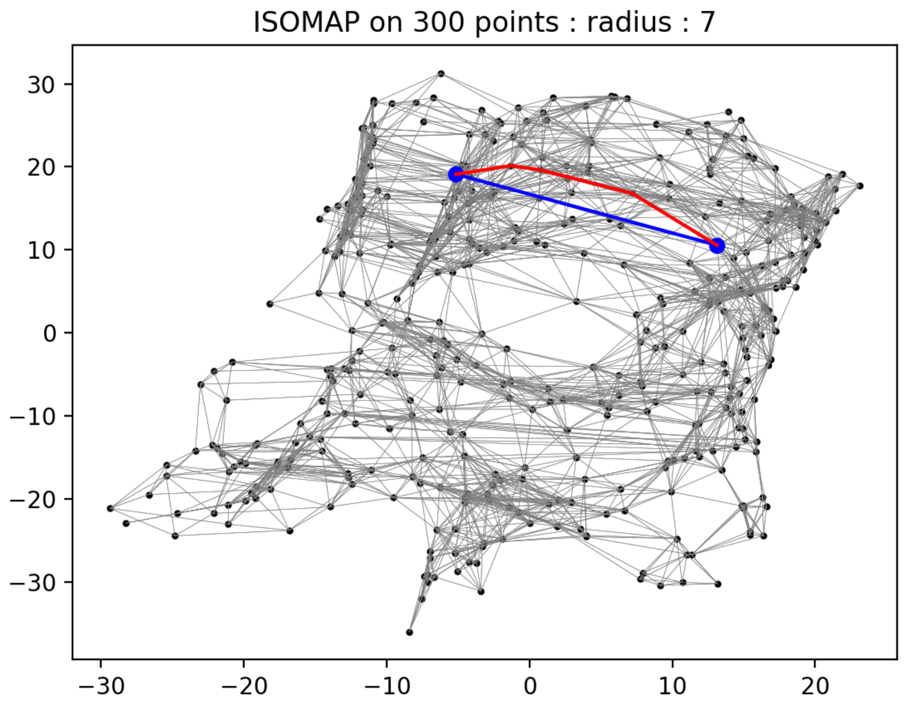


Figure 6: The 2−dimensional representation of the 300 points of Swiss roll dataset with ISOMAP

A Algorithms

Algorithm 1: Dijkstra's algorithm

```
Input : An undirected connected graph $G = (V, E)$,a weight function $w : E \longrightarrow \mathbb{R}^+$ and a source node
  $s \in V$ Output : A shortest path tree $T_s$ rooted at $s$.
$V_T := \{s\}$
$d[s] := 0$  une liste des distances
$E_T := \emptyset$
while $V_T \neq V$ do
    Let  $u \in V_T$  and  $v \in V \backslash v_T$  with  $d[u] + w(u, v)$  minimal
    $V_T = V_T \cup \{v\}$
    $E_T = E_T \cup \{(u, v)\}$
    $d[v] = d[u] + w(u, v)$
end while
return $(V_T, E_T)$
```

Algorithm 2: Bellman-Ford's algorithm

```
Input : An undirected connected graph $G = (V, E)$,a weight function $w : E \longrightarrow \mathbb{R}$ such that there exists
  no negative cycle and a source node $s \in V$
Output : An array $d[]$ containing the distances $d(s, v)$ from $s$ to all vertices $v \in V$.
$d[s] := 0$  une liste des distances
for $v \in V \backslash \{s\}$ do
    $d[v] := +\infty$
end for
for $n$=1 to $N$ do
    for $v \in V$ do
        for $u \in N^-(v)$ do
            if $d[u] + w(u, v) < d[v]$ then
                $d[v] = d[u] + w(u, v)$
            end if
        end for
    end for
end for
 - Detecting negative cycles :
for $v \in V$ do
    for $u \in N^-(v)$ do
        if $d[u] + w(u, v) < d[v]$ then
            Print("Negative cycle detected")
        end if
    end for
end for
return $d[.]$
```

Algorithm 3: Floyd-Warshall's algorithm

```
Input : An directed or undirected connected graph $G = (V, E)$,a weight function $w : E \longrightarrow \mathbb{R}$ such that
  there exists no negative cycle and a source node $s \in V$
Output : A matrix $d[][]$ containing the distances $d(i, j)$ between any two vertices $i$ and $j$.
$d[.][.]$  a distance matrix
for $i, j \in V\}$ do
    $d[i][j] = +\infty$
end for
for $i \in V\}$ do
    $d[i][i] = +\infty$
end for
for $(i, j) \in E\}$ do
    $d[i][j] = w(i, j)$
    $d[j][i] = w(i, j)$  for undirected graphs
end for
for $i$=1 to $n$ do
    for $j$=1 to $N$ do
        for $k$=1 to $N$ do
            if $d[i][j] > d[i][k] + d[k][j]$ then
                $d[i][j] = d[i][k] + d[k][j]$
            end if
        end for
    end for
end for
return $d[.][.]$
```

Algorithm 4: k-shortest path's algorithm

```
Input : An undirected connected graph $G = (V, E)$,a weight function $w : E \longrightarrow \mathbb{R}^+$, a source and target
  nodes $s, t \in V$ and a number $k \in \mathbb{N}$.
Output : k-shortest paths from $s$ to $t$. Parameters : we denote by :
$P_u$ a path from $s$ to $u$.
$PQ$ a heap containing paths (the key is the length).
$P$ the set of shortest paths from $s$ to $t$.
$Count[.]$ an array of length $n$ counting for each vertice the number of shortest paths from $s$ to this vertice
  already founded.
$P := \emptyset$
for $u \in V\}$ do
    $Count[u] = 0$
end for
$PQ.add(P_s = \{v\}, cost = 0)$
for $PQ$ is not empty and $Count[t] < k$ do
    $P_u, cost_u = PQ.pop()$
    $Count[u] = Count[u] + 1$
    if $u == t$ then
        $P = P \cup \{P_u\}$
    end if
    if $Count[u] \leqslant k$ then
        for $v \in N^+(u)$ do
            $P_v = P_u \cup \{v\}$
            $cost_v = cost + w(u, v)$
            $PQ.add(P_v, cost_v)$
        end for
    end if
end for
return $P$
```

Algorithm 5: ISOMAP's algorithm

```
Input : A set $Data$ of $N$ $n$-dimensional points (the data manifold) and a number $d$ representing the new
  reduced dimension.
Output : The set of the input points expressed in the new d-dimensional euclidian reduced space.
Parameters : we denote by :
$\epsilon$ a constant in $\mathbb{R}^+$ (chosen as an input).
— Neighborhood graph construction ————————————————————————
$V_N = Data_n$
$E_N = 0$
for $u \in V\}$ do
    for $v \in V\}$ do
        if $\|u - v\| \leqslant \epsilon$(Euclidian distance) then
            $E_N = E_N \cup \{v\}$
        end if
    end for
end for
$G_N = (V_N, E_N)$
— Shortest paths computation ————————————————————————
$d_G[.][.] = Floyd - Warshall(G_N)$  Geodesic distances matrix
— Dimensionality reduction ————————————————————————-
Run a MDS algorithm on $d_G[.][.]$ so as to compute $Data_d$ the set of $N$ $d$-dimensional points. return
  $Data_d$
```

References

[1] *Dijkstra's algo.* URL: http://www-m3.ma.tum.de/foswiki/pub/MN0506/WebHome/dijkstra.pdf.

[2] *Dijkstra's parallelization.* URL: https://people.mpi-inf.mpg.de/~mehlhorn/ftp/ParallelizationDijkstra.pdf.

[3] *Datasets for the (constrained) shortest path problem.* URL: http://people.brunel.ac.uk/~mastjjb/jeb/orlib/rcspinfo.html.

[4] *Dimensionaityreduction.* URL: https://www.cs.princeton.edu/courses/archive/fall13/cos323/notes/cos323_f13_lecture11_dim_red.pdf.

[5] *ISOMAP algorithm.* URL: https://web.mit.edu/cocosci/Papers/sci_reprint.pdf.

[6] *Geodesic geometry.* URL: http://web.mit.edu/deslab/pubs/geo.pdf.

[7] *Swiss roll data.* URL: http://web.mit.edu/cocosci/isomap/isomap.html.