IE440 FINAL22–Murat Ozturk 2019402093

Traveling Salesman Problem

The traveling salesman problem (TSP) is a classic problem in computer science and operations research. It involves finding the shortest possible route that visits a given set of locations and returns to the starting point.

Euclidean Traveling Salesman Problem

The Euclidean traveling salesman problem (ETSP) is a variant of the traveling salesman problem (TSP) where the distance between two locations is the Euclidean distance, which is the straight-line distance between the two points. The straight-line distance between two points is calculated using the Euclidean distance formula, which is based on the Pythagorean theorem.

ETSP using Self Organizing Map

Self-organizing maps (SOMs) are a type of artificial neural network that can be used for clustering data etc.

Self-organizing maps can also be used to solve the ETSP problem. If the algorithm works as intended, the weights of the nodes will represent the route passing through the points. It is important to note that the solution obtained using this approach may not necessarily be the shortest possible route.

The Kernels

First the kernels are defined.

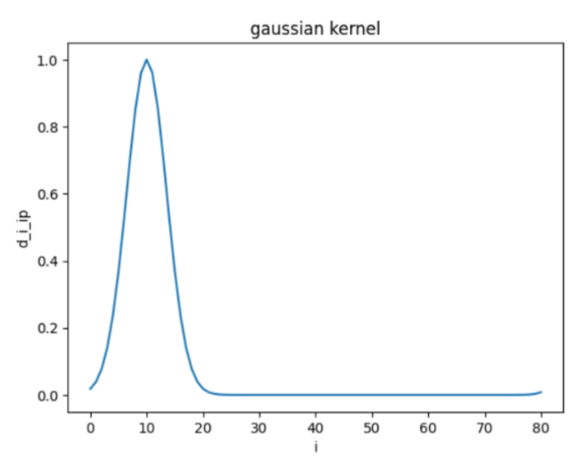
```
def gaussian_kernel(x, sigma):
    kernel = np.exp(-np.square(x)/np.square(sigma))
    return kernel
```

```
def elastic_band_kernel(x, sigma):
    kernel = np.where(x <= sigma, 1, gaussian_kernel(x, sigma))
    return kernel
```

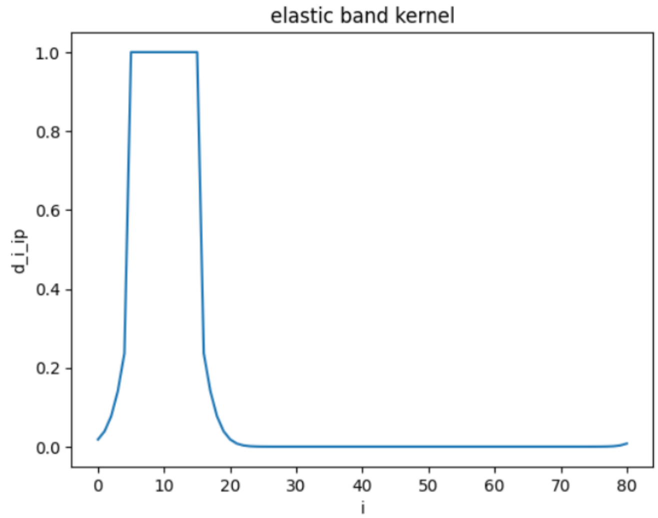
The input to these kernels will be diip. Below, is the example as the gaussian kernel.

$$d_{ii_p} = \min\{|i - i_p|, I - |i - i_p|\};$$
$$\Lambda^{(t)}(i, i_p) = \exp\left(-\frac{d_{ii_p}^2}{(\sigma^{(t)})^2}\right);$$

In the algorithm to be provided, the diip values will be given as a vector d_i which contains values d_0_ip, d_1_ip, d_2_ip, … d_M_ip. The output of the kernels will be like the plot, where the peak is the best matching unit's index. In the figure, there are 81 neurons.



For the elastic band kernel, the plot will be like figure below.



The Algorithm

The documentation of the parameters:

```
def SOM_ETSP(X, alpha=0.8, M=None, sigma=None, gama=0.99, beta=0.95, kernel_function=gaussian_kernel, max_iter=3000, seed=None, track=False):
    ''' ETSP using SOM (Euclidean Traveling Salesman Problem using Self-Organizing Map).
    Parameters
    ----------
    X : array-like, shape = [n, dim]
        Input data, n is the number of cities, dim is the dimension of the problem.
    M : int
        Number of neurons.
    alpha : float
        Learning rate.
    sigma : float
        Neighborhood radius.
    gama : float
        Reduction factor of learning rate.
    beta : float
        Reduction factor of neighborhood radius.
    kernel_function : function
        Kernel function to use. Default is gaussian_kernel.
    max_iter : int
        Maximum number of iterations.
    seed : int
        Random seed.
    track : bool
        If True, print the iteration number and plot the results.
    Returns
    -------
    W : array-like, shape = [M, dim]
        Weight matrix. This is the solution of ETSP.
        It should give the route that visits all the cities in the shortest distance.
        If M>n, some of the neurons will be used to represent the same city.
    '''
```

How the algorithm works:

First some values are initialized.
The default for M is number of cities.
Default for sigma is M/2.
Weights are initialized randomly between 0 and 1
W_prev holds, previous weights. This will be explained later.

```
np.random.seed(seed)

# Initialization
n, dim = X.shape
if(M is None): M = n
if(sigma is None): sigma = M/2
W = np.random.rand(M, dim)
W_prev = W.copy()
```

Then the iterations for loop:

```
for it in range(max_iter):
    r_order = np.random.permutation(n)  # Random order of cities
```

Random permutation is created at length of the cities.
r_order (random _ order)

Iterate all cities, for loop:

```
    for p in r_order:
        # Find the best matching unit
        distances = np.sum(np.power(X[p] - W, 2), axis=1)
        i_p = np.argmin(distances)
```

For each city, using the order in r_order, the distances to each neuron weights are calculated. Then the smallest distance, thus the best matching unit (BMU) is found. ‘i_p’ is the index of the BMU.

```
        # Update the weights
        order_distance = np.abs(np.arange(M) - i_p)
        d_i = np.minimum(order_distance, M - order_distance)
        kernel = kernel_function(d_i, sigma)
        kernel = kernel.reshape(-1, 1)      # Reshape to make it broadcastable
        W = W + alpha * kernel * (X[p] - W)
```

Order_distance is diip. d_i vector is calculated. The kernel vector is calculated using the chosen kernel function. Then 1 dimensional kernel vector is reshaped into (M, 1) 2 dimension to be broadcasted. Weights are updated accordingly.

End for loop, p in r_order

```
    # Update the learning rate and neighborhood radius
    alpha = alpha * gama
    sigma = sigma * beta
```

The learning rate and neighborhood radius are updated.

```
    # Plot the results every 5% of the max iterations
    if(track and it % (max_iter/20) == 0):
        print(it)
        plot_cities(X, W, it)
```

If track is set to True, then the process of training can be tracked with plots.

```
    # Break if the change in W's are all too small or learning rate is too small
    if np.all(np.sum(np.power(W_prev - W, 2), axis=1) < 1e-5) or alpha < 1e-5:
        break

    # Update W_prev
    W_prev = W.copy()
```

Break for loop if the changes in all weights are too small, or the learning rate is too small.
If not, record the current weights as W_prev to compare in the next iteration.

End for loop, it in range(max_iter)
Finally return the results.

All the algorithm in one picture:

```python
np.random.seed(seed)

# Initialization
n, dim = X.shape
if(M is None): M = n
if(sigma is None): sigma = M/2
W = np.random.rand(M, dim)
W_prev = W.copy()

# Iteration
for it in range(max_iter):
    r_order = np.random.permutation(n)  # Random order of cities

    for p in r_order:
        # Find the best matching unit
        distances = np.sum(np.power(X[p] - W, 2), axis=1)
        i_p = np.argmin(distances)

        # Update the weights
        order_distance = np.abs(np.arange(M) - i_p)
        d_i = np.minimum(order_distance, M - order_distance)
        kernel = kernel_function(d_i, sigma)
        kernel = kernel.reshape(-1, 1)      # Reshape to make it broadcastable
        W = W + alpha * kernel * (X[p] - W)

    # Update the learning rate and neighborhood radius
    alpha = alpha * gama
    sigma = sigma * beta

    # Plot the results every 5% of the max iterations
    if(track and it % (max_iter/20) == 0):
        print(it)
        plot_cities(X, W, it)

    # Break if the change in W's are all too small or learning rate is too small
    if np.all(np.sum(np.power(W_prev - W, 2), axis=1) < 1e-5) or alpha < 1e-5:
        break

    # Update W_prev
    W_prev = W.copy()

# Plot the final results
print('Stopped at iteration:', it)
print('Route length:', calc_route_length(W))
plot_cities(X, W, it)

return W
```

Outputs

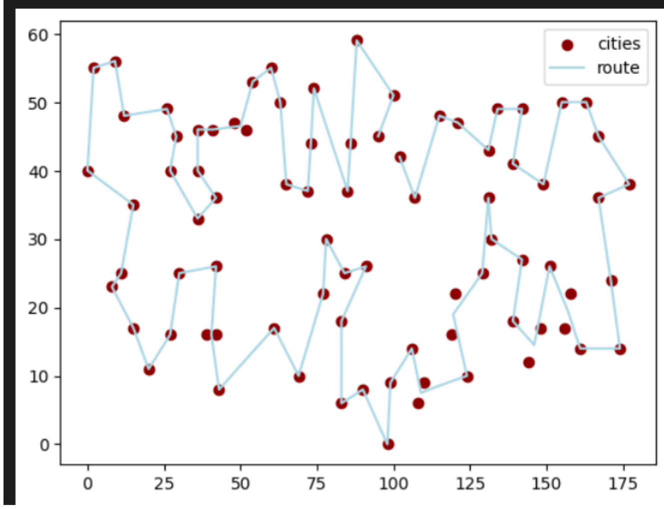
Gaussian Kernel

M = n

```
y = SOM_ETSP(X_cities, kernel_function=gaussian_kernel, alpha=1, gama=0.997, sigma=5, beta=0.982, M=81, seed=0, max_iter=10000)
```

✓ 1.9s

```
Stopped at iteration: 1130
Route length: 724.7915781461614
```



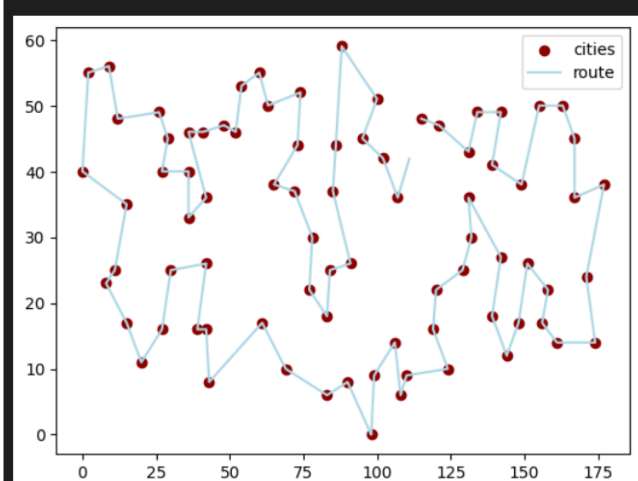
The result isn't satisfying. There are some cities that are not covered by the route.

M = 2n

```
y = SOM_ETSP(X_cities, kernel_function=gaussian_kernel, alpha=0.95, gama=0.997, sigma=81, beta=0.95, M=81*2, seed=0, max_iter=10000)
```

✓ 0.3s

```
Stopped at iteration: 107
Route length: 731.4324862218724
```



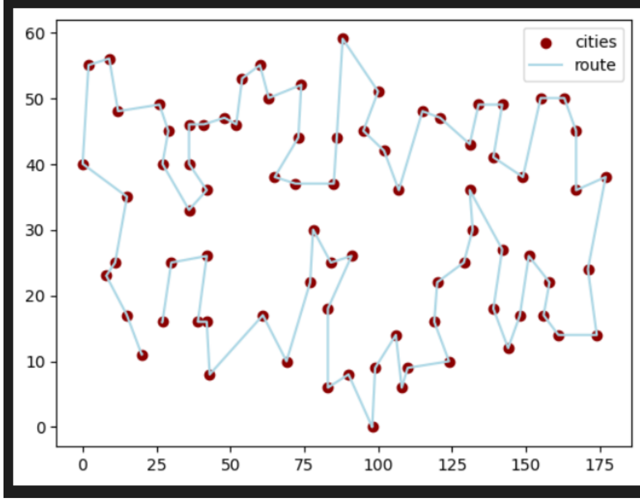
All cities are covered, this can be used as a solution.

M = 3n

```
y = SOM_ETSP(X_cities, kernel_function=gaussian_kernel, alpha=0.75, gama=0.999, sigma=81*3, beta=0.975, M=81*3, seed=0, max_iter=500)
```

✓ 0.6s

```
Stopped at iteration: 244
Route length: 736.3479208756254
```



All cities are covered, this can be used as a solution.

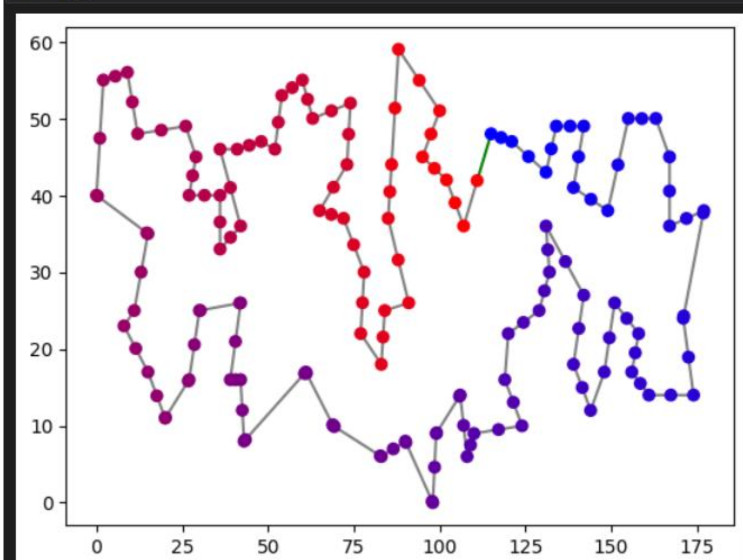
Best Solution

```
kernel_function=gaussian_kernel, alpha=0.95, gama=0.997, sigma=81, beta=0.95, M=81*2, seed=0
```

Route starts from green line, blue to red.

```
plot_route(y_best)
```

✓ 0.9s




```
Route length: 731.4324862218724
```

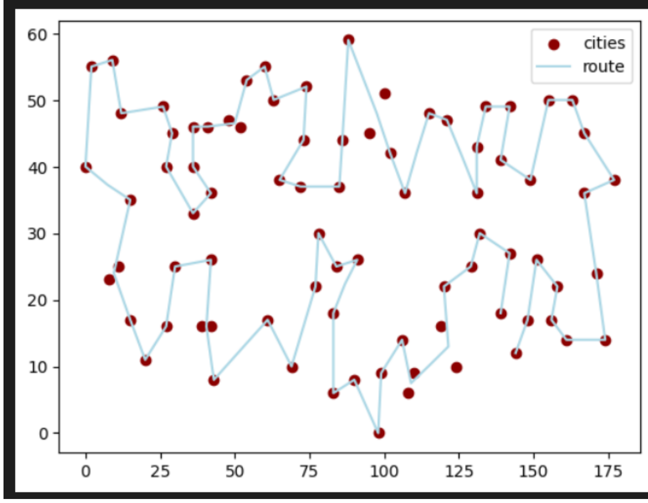
Elastic Band Kernel

M = n

```
y = SOM_ETSP(X_cities, kernel_function=elastic_band_kernel, alpha=0.75, gama=0.99, sigma=81, beta=0.95, M=81, seed=0, max_iter=10000)
```

✓ 0.6s

```
Stopped at iteration: 319
Route length: 713.3873220161616
```



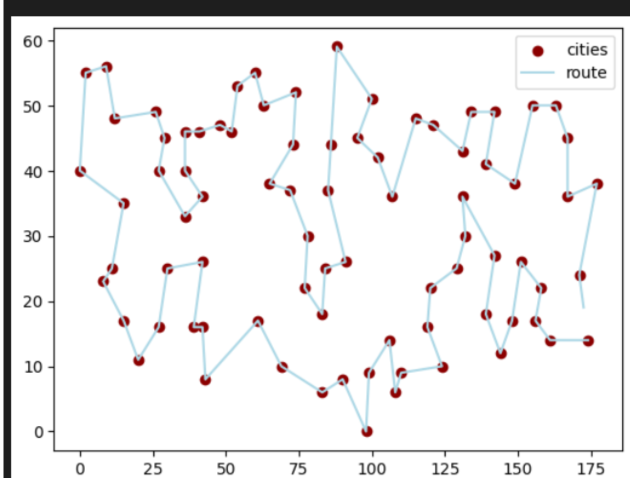
The result isn't satisfying. There are some cities that are not covered by the route.

M = 2n

```
y = SOM_ETSP(X_cities, kernel_function=elastic_band_kernel, alpha=0.75, gama=0.995, sigma=10, beta=0.985, M=81*2, seed=0, max_iter=10000)
```

✓ 0.5s

```
Stopped at iteration: 216
Route length: 728.3027556322063
```



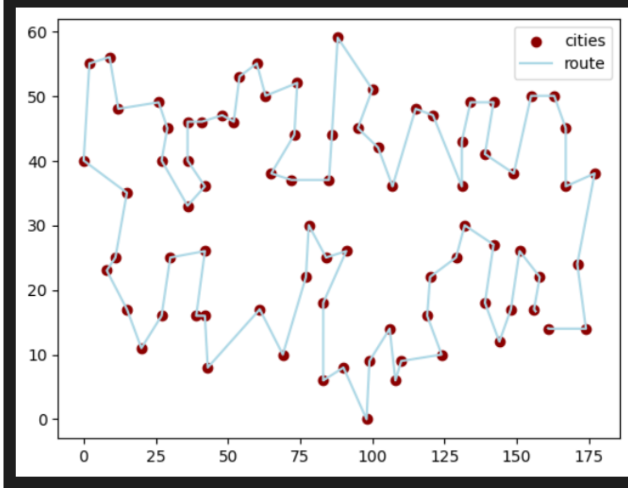
All cities are covered, this can be used as a solution.

M = 3n

```
y = SOM_ETSP(X_cities, kernel_function=elastic_band_kernel, alpha=0.75, gama=0.99, sigma=25, beta=0.98, M=81*3, seed=0, max_iter=10000)
```

✓ 0.5s

```
Stopped at iteration: 190
Route length: 737.4806692775311
```



All cities are covered, this can be used as a solution.

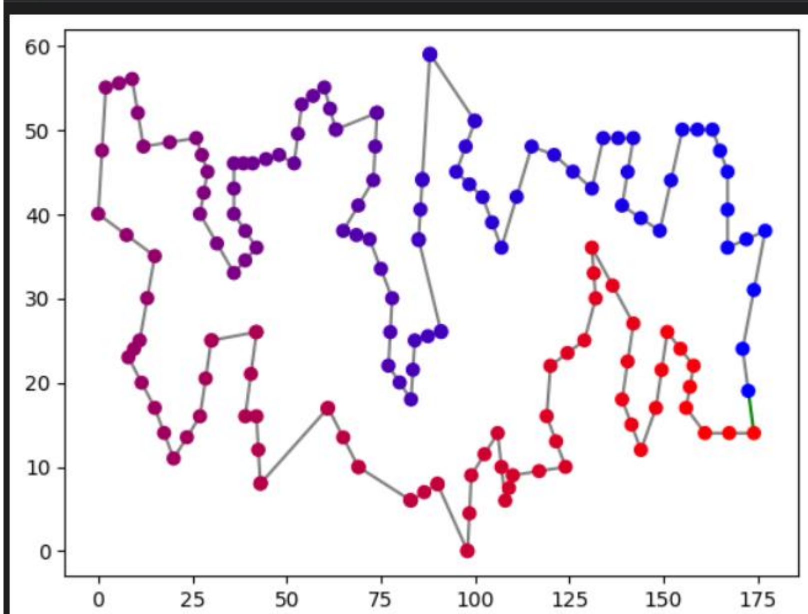
Best Solution

```
kernel_function=elastic_band_kernel, alpha=0.75, gama=0.995, sigma=10, beta=0.985, M=81*2, seed=0
```

Route starts from green line, blue to red.

```
plot_route(y_best)
```

✓ 0.7s




```
Route length: 728.3027556322063
```