Report

IT3105 Assignment 3

Solving the travelling salesman problem using Kohonen SOM

Stefan Sobczyszyn Borg

Overview description of system

The system consists of four different java classes and a .css file.

- Class NodePoint.java: Used to represent the cities of the input country. Each city has an own NodePoint instance where its coordinates and the BMU is stored.
- Class DynamicNodePoint.java: Used to represent the nodes of the elastic circle. Each node has an own DynamicNodePoint instance where its coordinates, closest city and distance to this city is stored.
- Class ScatterChartSample.java: Used for the visual representation of data from the SOMTravelingSalesMan class. The data are stored in separate files (start, mid, end) on disk and read by the class instance. JavaFX is used for the representation. There are three series on each chart. One representing the cities (red stars), one representing the circle of nodes (yellow dots connected by black lines) and the final one to connect the first and last element to complete the circle (green dots connected by a black line).
- Scatterchart.css: Styles used for the series representation in ScatterChartSample.java
- Class SOMTravelingSalesMan: Main class of the system containing most of the algorithm implementation. The main method firstly initiates the algorithm by asking the user for input. The coordinates are fetched from the source for later use, and decay types are set. An instance of the class is created and the nodes are produced randomly, lying between the highest and lowest coordinate values. The algorithm is now ready to start improving the solution. For each iteration a random city is selected. Its best matching unit is found and the neighborhood is updated. Both neighborhood radius and learning rates are adjusted in accordance to user input. If the algorithm is on the correct iteration (specified by user), the length of the shortest path so far is computed and

printed. The algorithm ends after it has completed the number of iterations specified in the code.

Some thorough description of steps in the algorithm follows.

- **Initiating and representing node circle:** The node circle is represented as a LinkedList<DynamicNodePoint>. Each node is created randomly in range of the lowest and highest city-coordinate from the data set. The circle is formed by adjacency in the list, with the last entry connecting to the first to complete the circle.
- **Finding the best matching unit(method: findBMU):** To find the best matching unit for a city all nodes are iterated and the Euclidean distance between the node and the city is calculated. The method returns the BMUs index in the nodes-list.
- **Updating the neighborhood and nodes(method updateNeighborhood and updateNode):** The index of the BMU and the city (NodePoint) are entered as method input. The method iterates over the neighborhood radius and updates all nodes in the following way:

  $$BMU = BMU + f(BMU) * l * (BMU - city) \qquad ,$$

  Where BMU is the coordinate vector of the best matching unit, f(BMU) is the neighborhood function, l is the learning rate and city is the coordinate vector of the current city.
- **Calculating neighborhood function(method: neighborhoodFunction):** The neighborhood function is a Gaussian function. Its input is the distance of the current node from the BMU. The value returned ranges from 0 to 1.
- **Finding shortest path:** Each city has its BMU unit stored. To find the shortest path the cities are sorted based on position in the ring. Cities connected to the same node are randomly sorted. To find the shortest path simply iterate the sorted list and calculate the distance between the cities.

Experiences tuning the system

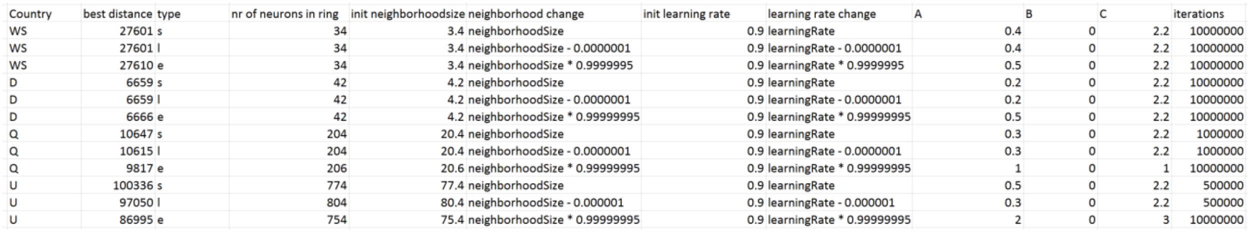
Tuning the parameters is critical for the performance of the algorithm. In my experience the most critical parameter is the circle size. While testing I found that the algorithm performs best when the circle size is about 10% larger than the number of cities for the smallest maps, and between 5-10% for the biggest maps. With greater circle size comes the cost of greater computation time for each iteration, so keeping the circle size as small as possible is desirable. It seems that having a circle with fewer nodes than cities on the map (about -5-20%) can give better performance for the bigger maps. The explanation for this can be that where the city density is high, cities will connect to the same node giving a good approximation for the shortest path.

Setting the neighborhood radius was also an important factor. Too high or too low neighborhoods would decrease the performance and precision drastically. During testing, I found, as it also says in the compendium, that having a neighborhood at approximately one tenth of the total circle size gives the best performance.

The constants in the neighborhood function where of a lesser importance than the above mentioned. However, tuning the constants correctly would decrease the time for the algorithm to find a good solution. It seemed that scaling the neighborhood weighting by a factor of 0.5 increased the performance. This might mean that the learning rate should be lower, given the relation between these two parameters. However, I found that setting the learning rate high and scaling the neighbor weight gives a better performance in the case where linear or exponential decay is used.

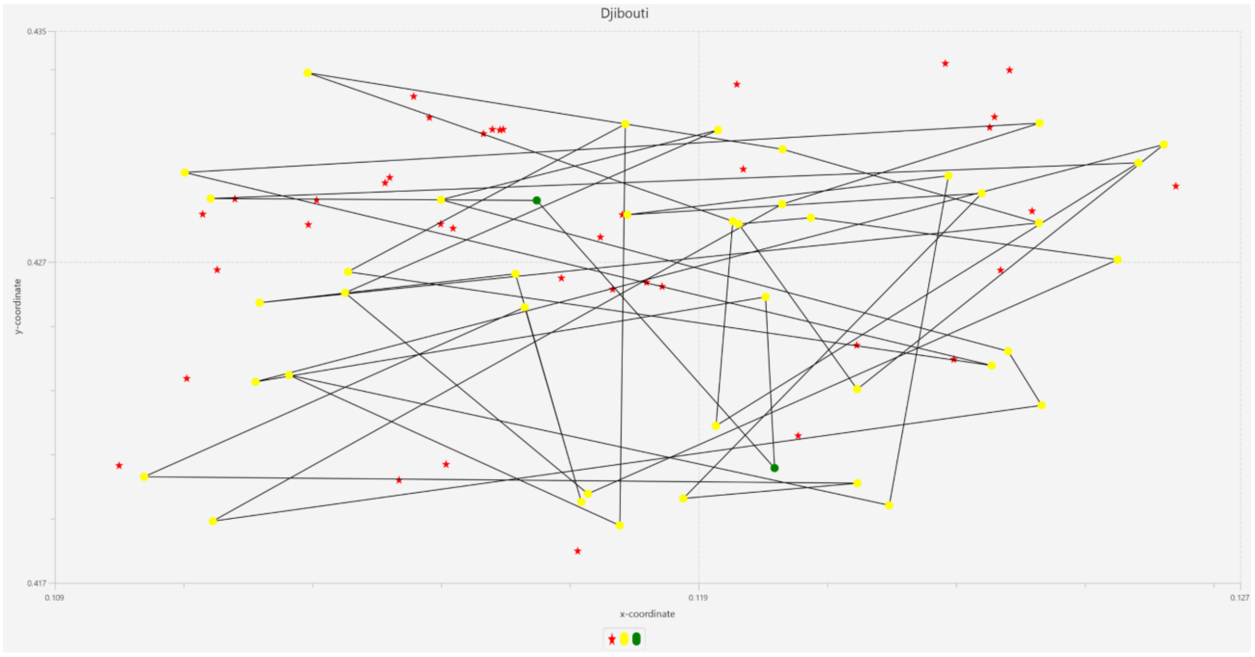
When it comes to static vs. linear vs. exponential decay types I found that the choice taken is not too critical for the performance. Generally, the algorithm would converge a bit faster with exponential decay types, but after many iterations the other two would catch up giving an equally good solution. However, for the larger maps it seemed that the static type started falling behind. I suspect that with a larger data set the difference in performance of the decay types will start to become significantly larger. Also, linear decay would generally converge faster than static.

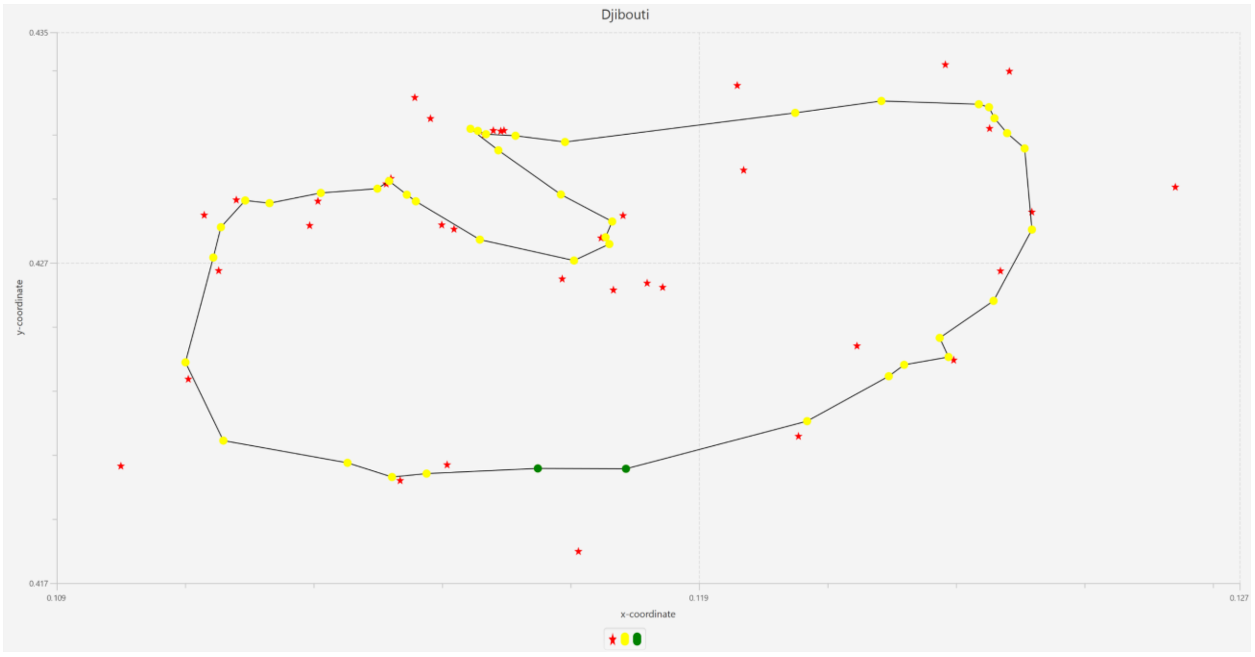
Table of parameters and distance after testing:

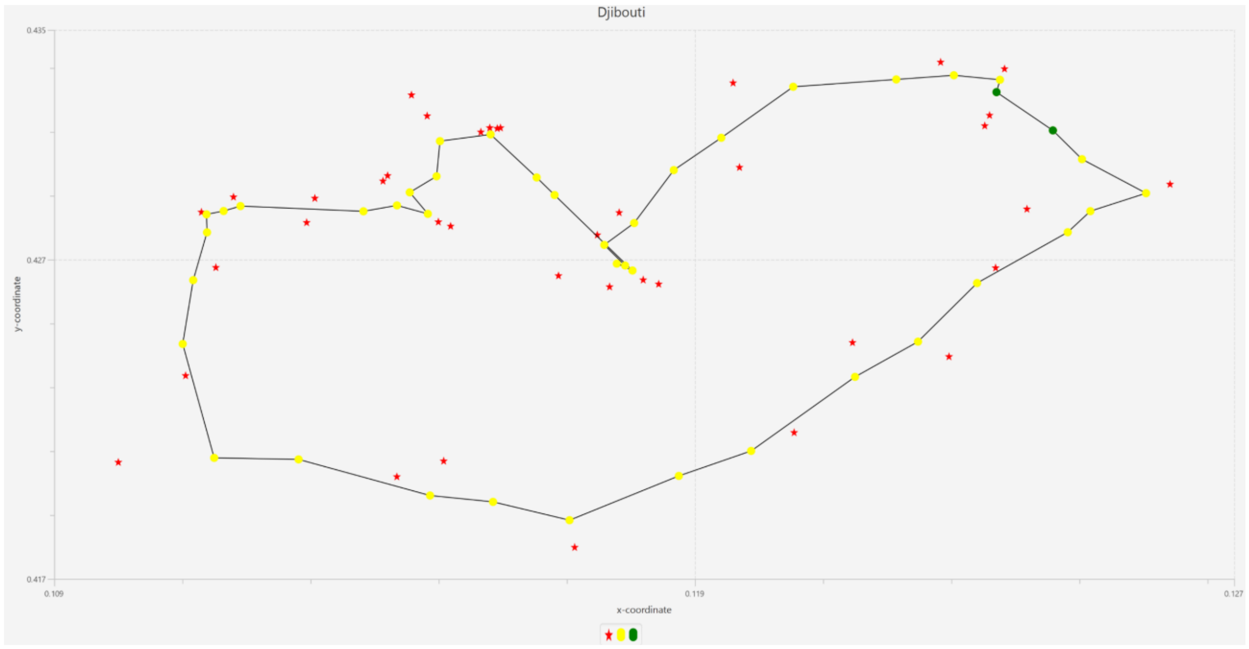


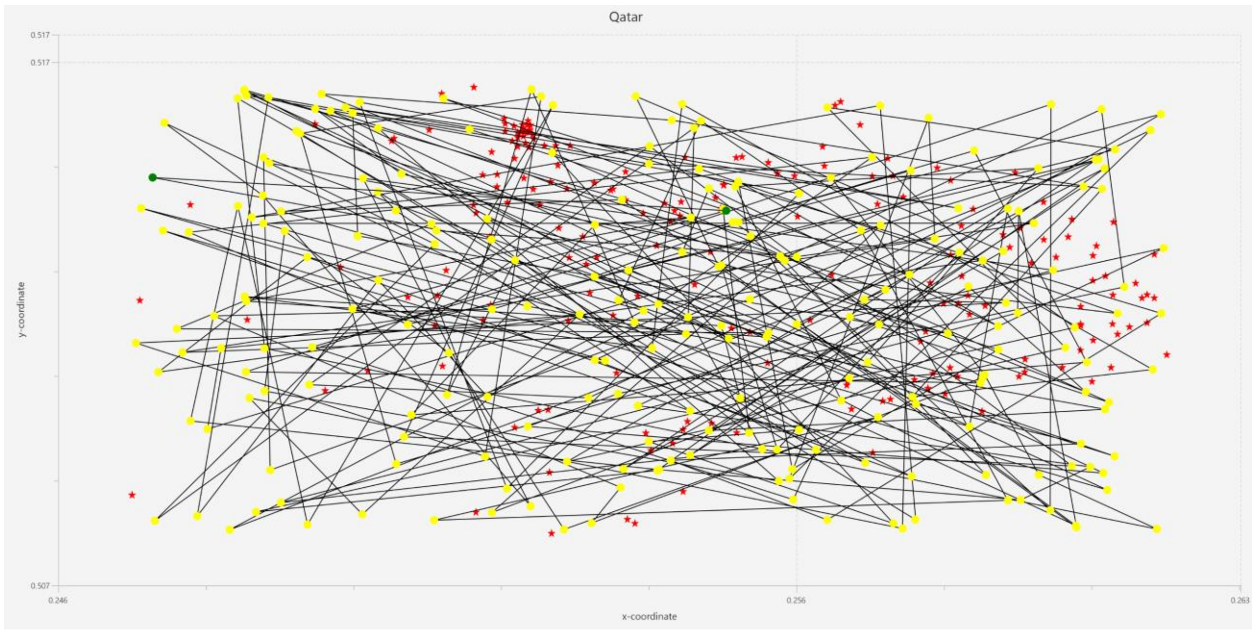
Country	best distance	type	nr of neurons in ring	init neighborhoodsize	neighborhood change	init learning rate	learning rate change	A	B	C	iterations
WS	27601	s	34	3.4	neighborhoodSize	0.9	learningRate	0.4	0	2.2	10000000
WS	27601	l	34	3.4	neighborhoodSize - 0.0000001	0.9	learningRate - 0.0000001	0.4	0	2.2	10000000
WS	27610	e	34	3.4	neighborhoodSize * 0.9999995	0.9	learningRate * 0.9999995	0.5	0	2.2	10000000
D	6659	s	42	4.2	neighborhoodSize	0.9	learningRate	0.2	0	2.2	10000000
D	6659	l	42	4.2	neighborhoodSize - 0.0000001	0.9	learningRate - 0.0000001	0.2	0	2.2	10000000
D	6666	e	42	4.2	neighborhoodSize * 0.99999995	0.9	learningRate * 0.99999995	0.5	0	2.2	10000000
Q	10647	s	204	20.4	neighborhoodSize	0.9	learningRate	0.3	0	2.2	1000000
Q	10615	l	204	20.4	neighborhoodSize - 0.0000001	0.9	learningRate - 0.0000001	0.3	0	2.2	1000000
Q	9817	e	206	20.6	neighborhoodSize * 0.99999995	0.9	learningRate * 0.99999995	1	0	1	10000000
U	100336	s	774	77.4	neighborhoodSize	0.9	learningRate	0.5	0	2.2	500000
U	97050	l	804	80.4	neighborhoodSize - 0.000001	0.9	learningRate - 0.000001	0.3	0	2.2	500000
U	86995	e	754	75.4	neighborhoodSize * 0.99999995	0.9	learningRate * 0.99999995	2	0	3	10000000

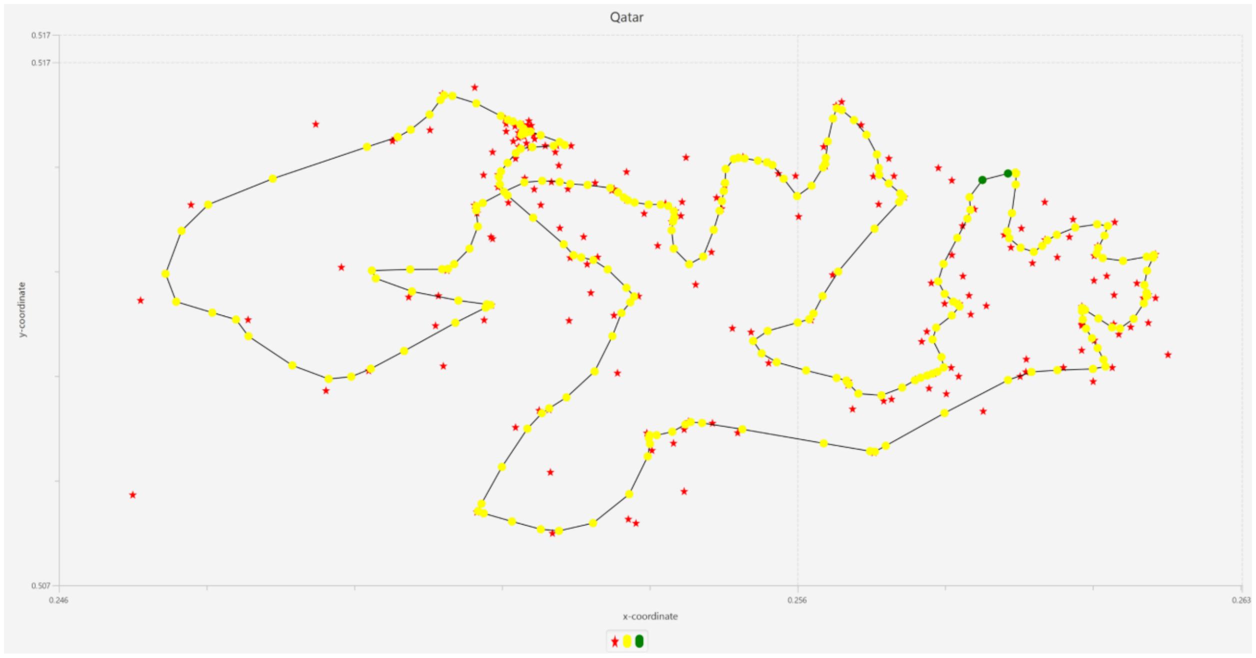
Diagrams (start, mid, end):

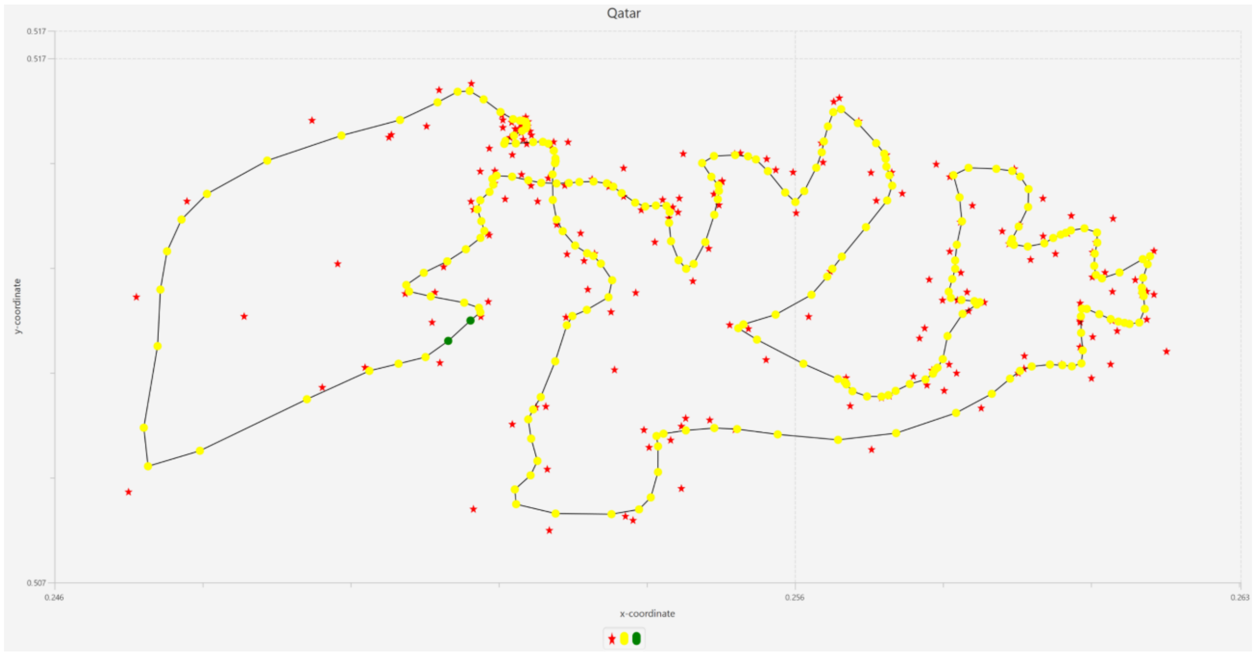


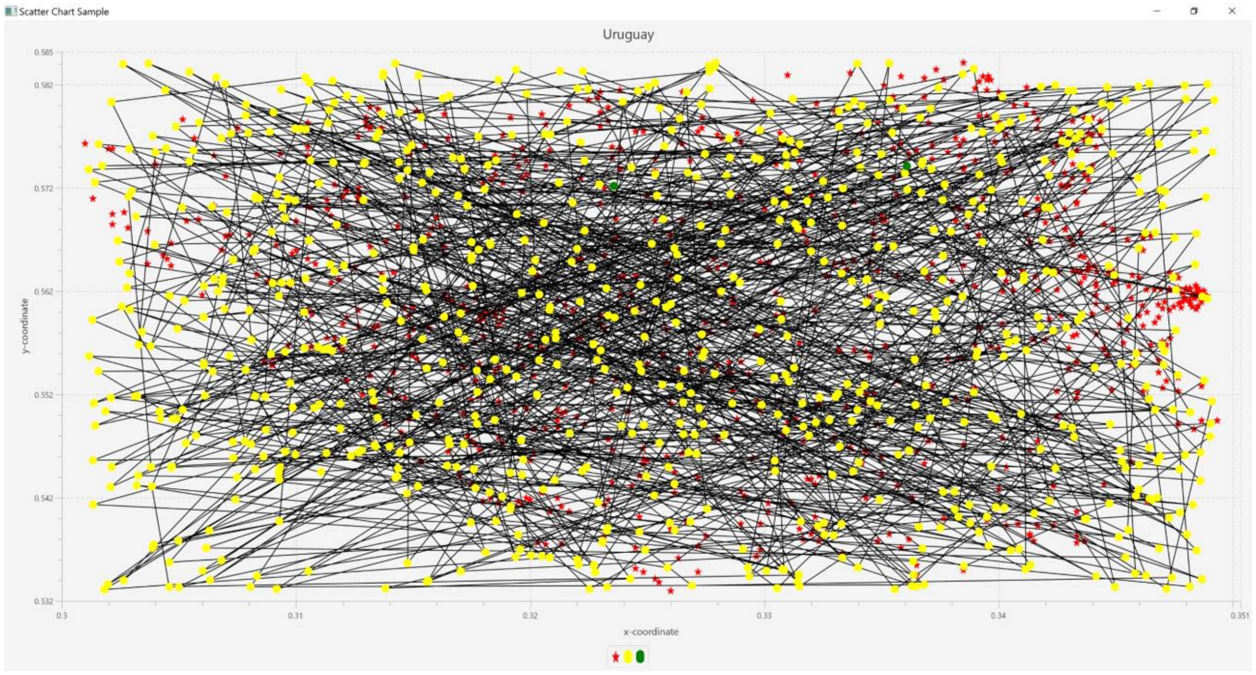


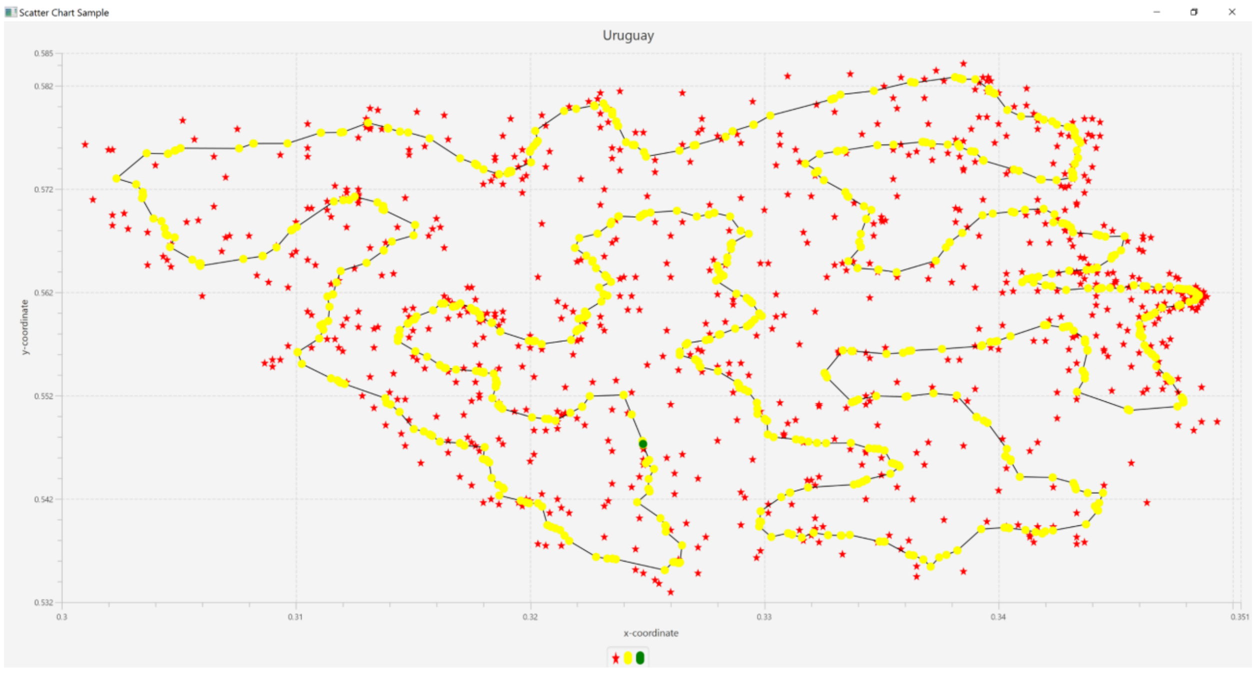

Djibouti
0.435
0.427
0.417
y-coordinate
0.109
0.119
0.127
x-coordinate

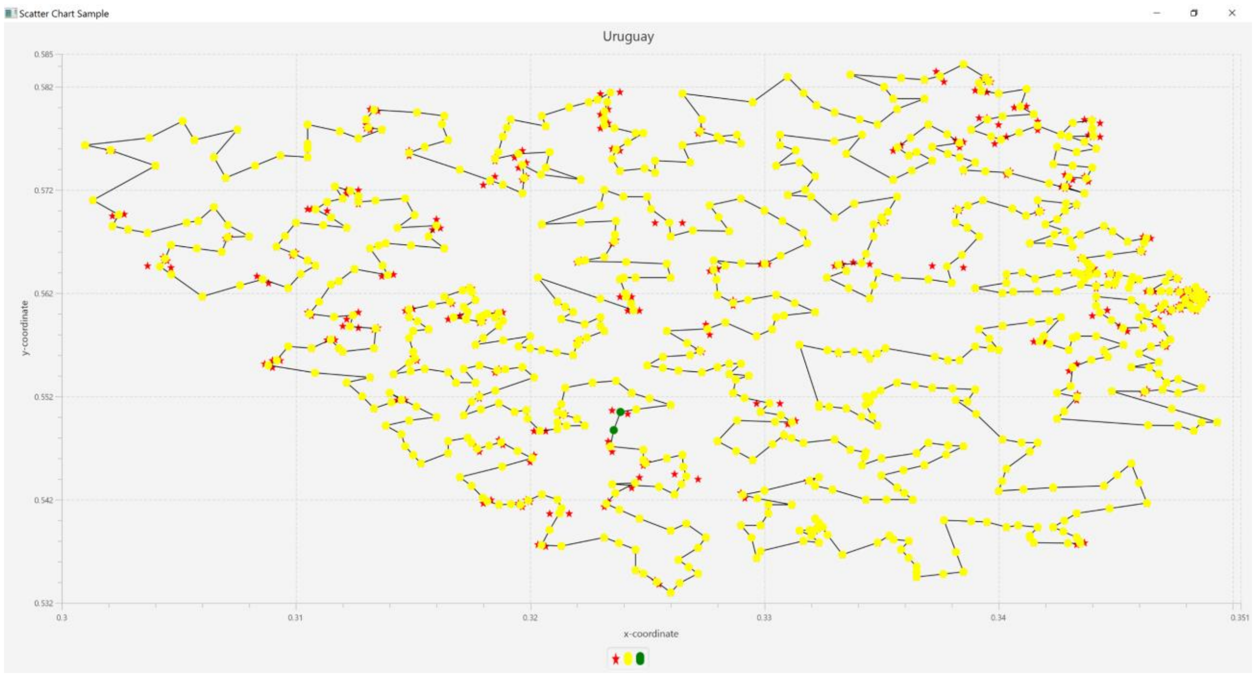

Qatar
0.517
0.517
0.507
y-coordinate
0.246
0.256
0.263
x-coordinate

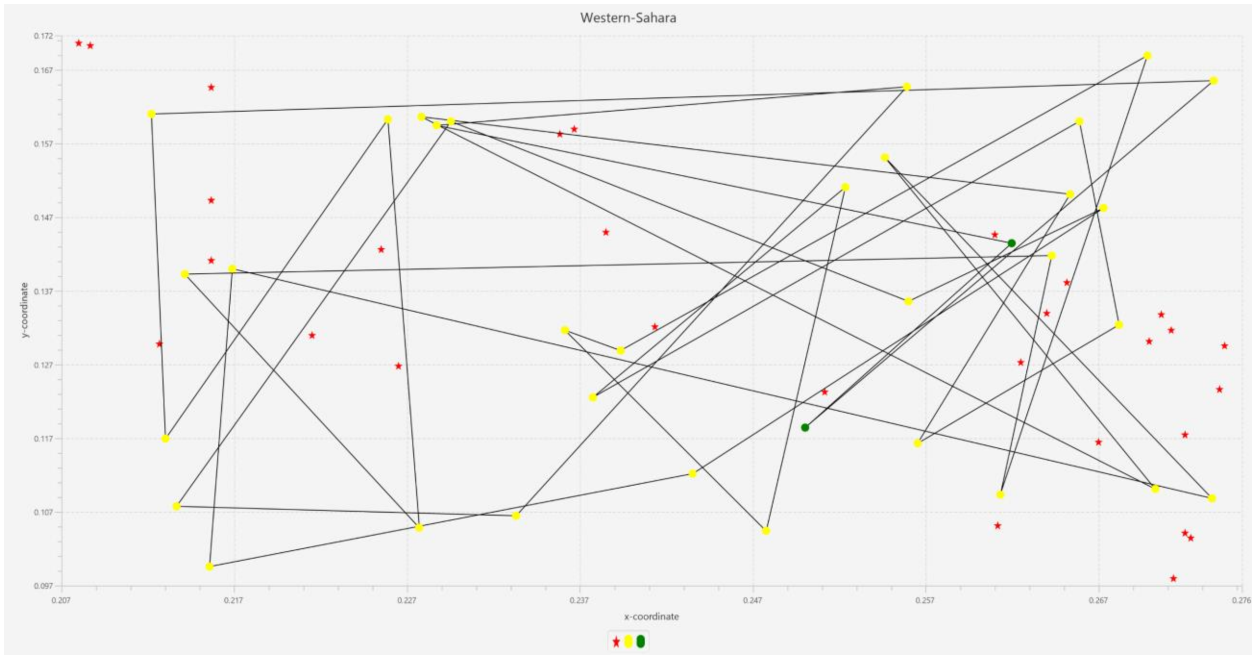

Qatar
0.517
0.517
0.507
y-coordinate
0.246
0.256
0.263
x-coordinate

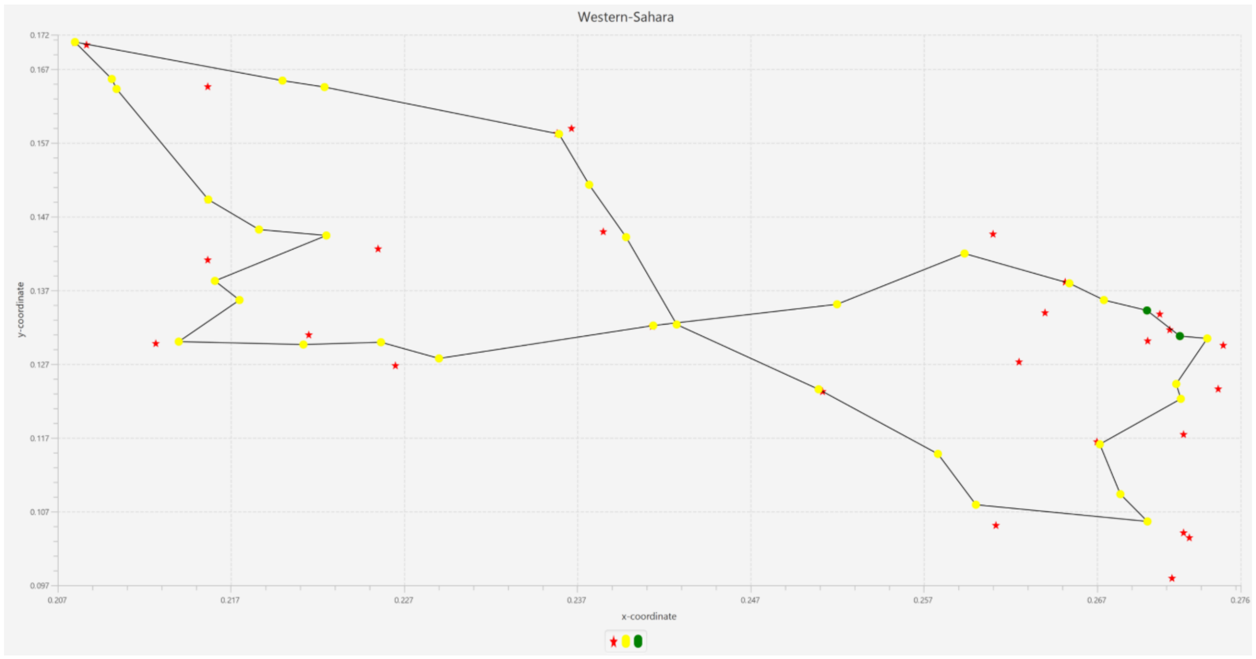

Qatar
0.517
0.517
0.507
y-coordinate
0.246
0.256
0.263
x-coordinate

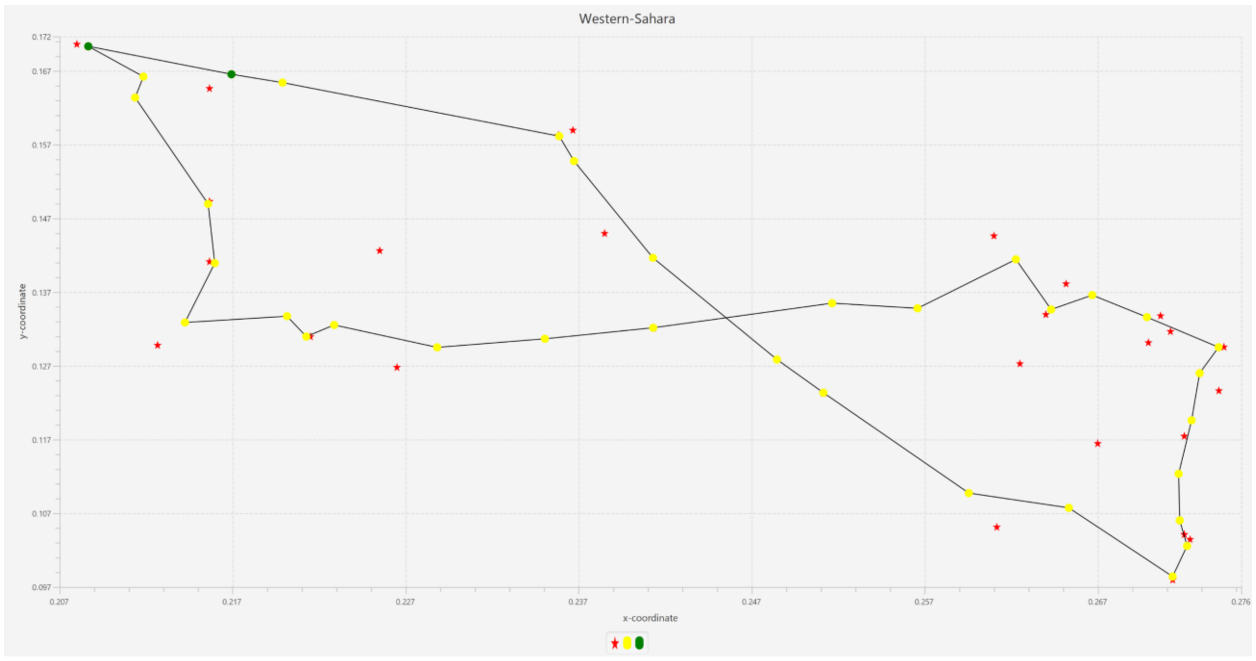

Scatter Chart Sample
Uruguay
y-coordinate
x-coordinate
0.585
0.582
0.572
0.562
0.552
0.542
0.532
0.3
0.31
0.32
0.33
0.34
0.351


Scatter Chart Sample
Uruguay
y-coordinate
x-coordinate
0.585
0.582
0.572
0.562
0.552
0.542
0.532
0.3
0.31
0.32
0.33
0.34
0.351


Scatter Chart Sample
Uruguay
y-coordinate
x-coordinate
0.585
0.582
0.572
0.562
0.552
0.542
0.532
0.3
0.31
0.32
0.33
0.34
0.351


Western-Sahara
y-coordinate
x-coordinate
0.172
0.167
0.157
0.147
0.137
0.127
0.117
0.107
0.097
0.207
0.217
0.227
0.237
0.247
0.257
0.267
0.276


Western-Sahara
y-coordinate
x-coordinate
0.172
0.167
0.157
0.147
0.137
0.127
0.117
0.107
0.097
0.207
0.217
0.227
0.237
0.247
0.257
0.267
0.276


Western-Sahara
y-coordinate
x-coordinate
0.172
0.167
0.157
0.147
0.137
0.127
0.117
0.107
0.097
0.207
0.217
0.227
0.237
0.247
0.257
0.267
0.276