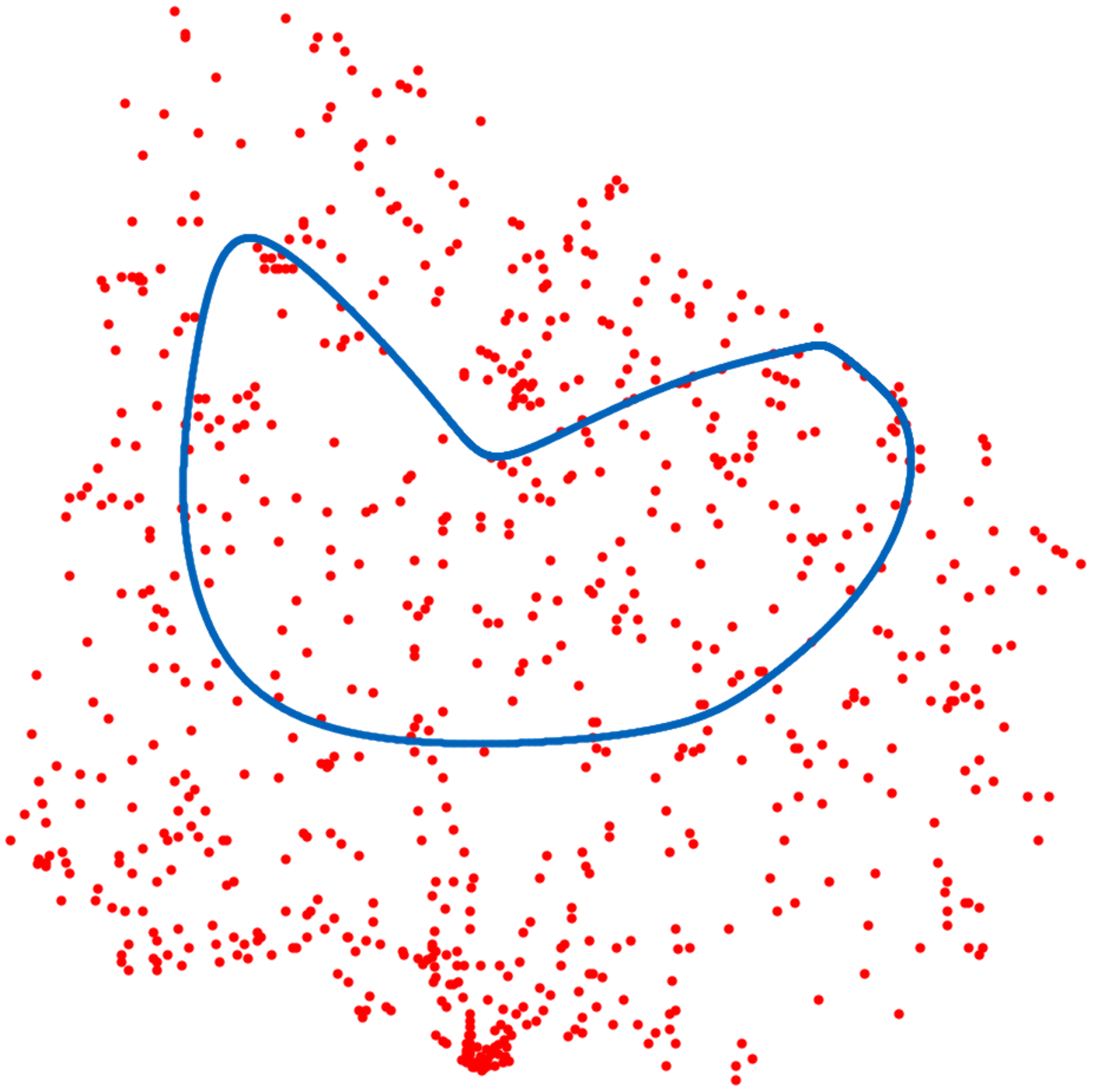
Using Self-Organizing Maps to solve the Traveling Salesman Problem

The Traveling Salesman Problem is a well known challenge in Computer Science: it consists on finding the shortest route possible that traverses all cities in a given map only once. Although its simple explanation, this problem is, indeed, NP-Complete. This implies that the difficulty to solve it increases rapidly with the number of cities, and we do not know in fact a general solution that solves the problem. For that reason, we currently consider that any method able to find a sub-optimal solution is generally good enough (we cannot verify if the solution returned is the optimal one most of the times).

To solve it, we can try to apply a modification of the Self-Organizing Map (SOM) technique. Let us take a look at what this technique consists, and then apply it to the TSP once we understand it better.



Note (2018-02-01): You can also read this post in Chinese, translated by Yibing Du.

# Some insight on Self-Organizing Maps

The original paper released by Teuvo Kohonen in 1998[1] consists on a brief, masterful description of the technique. In there, it is explained that a *self-organizing map* is described as an (usually two-dimensional) grid of nodes, inspired in a neural network. Closely related to the map, is the idea of the *model*, that is, the real world observation the map is trying to represent. The purpose of the technique is to represent the model with a lower number of dimensions, while maintaining the relations of similarity of the nodes contained in it.

To capture this similarity, the nodes in the map are spatially organized to be closer the more similar they are with each other. For that reason, SOM are a great way for pattern visualization and organization of data. To obtain this structure, the map is applied a regression operation to modify the nodes position in order update the nodes, one element from the model (e) at a time. The expression used for the regression is:

$$n_{t+1} = n_t + h(w_e) \cdot \Delta(e, n_t)$$

This implies that the position of the node $n$ is updated adding the distance from it to the given element, multiplied by the neighborhood factor of the winner neuron, w_e. The *winner* of an element is the more similar node in the map to it, usually measured by the closer node using the Euclidean distance (although it is possible to use a different similarity measure if appropriate).

On the other side, the *neighborhood* is defined as a convolution-like kernel for the map around the winner. Doing this, we are able to update the winner and the neurons nearby closer to the element, obtaining a soft and proportional result. The function is usually defined as a Gaussian distribution, but other implementations are as well. One worth mentioning is a bubble neighborhood, that updates the neurons that are within a radius of the winner (based on a discrete Kronecker delta function), which is the simplest neighborhood function possible.

# Modifying the technique

To use the network to solve the TSP, the main concept to understand is how to modify the neighborhood function. If instead of a grid we declare a *circular array of neurons*, each node will only be conscious of the neurons in front of and behind it. That is, the inner similarity will work just in one dimension. Making this slight modification, the self-organizing map will behave as an elastic ring, getting closer to the cities but trying to minimize the perimeter of it thanks to the neighborhood function.

Although this modification is the main idea behind the technique, it will not work as is: the algorithm will hardly converge any of the times. To ensure the convergence of it, we can include a learning rate, α, to control the exploration and exploitation of the algorithm. To obtain high exploration first, and high exploitation after that in the execution, we must include a *decay* in both the neighborhood function and the learning rate. Decaying the learning rate will ensure less aggressive displacement of the neurons around the model, and decaying the neighborhood will result in a more moderate exploitation of the local minima of each part of the model. Then, our regression can be expressed as:

$$n_{t+1} = n_t + \alpha_t \cdot g(w_e, h_t) \cdot \Delta(e, n_t)$$

Where α is the learning rate at a given time, and (g) is the Gaussian function centered in a winner and with a neighborhood dispersion of h. The decay function consists on simply multiplying the two given discounts, γ, for the learning rate and the neighborhood distance.

$$\alpha_{t+1} = \gamma_\alpha \cdot \alpha_t, \;\; h_{t+1} = \gamma_h \cdot h_t$$

This expression is indeed quite similar to that of Q-Learning, and the convergence is search in a similar fashion to this technique. Decaying the parameters can be useful in unsupervised learning tasks like the aforementioned ones. It is also similar to the functioning of the Learning Vector Quantization technique, also developed by Teuvo Kohonen.

Finally, to obtain the route from the SOM, it is only necessary to associate a city with its winner neuron, traverse the ring starting from any point and sort the cities by order of appearance of their winner neuron in the ring. If several cities map to the same neuron, it is because the order of traversing such cities have not been contemplated by the SOM (due to lack of relevance for the final distance or because of not enough precision). In that case, any possible ordered can be considered for such cities.

# Implementing and testing the SOM

For the task, an implementation of the previously explained technique is provided in Python 3. It is able to parse and load any 2D instance problem modelled as a `TSPLIB` file and run the regression to obtain the shortest route. This format is chosen because for the testing and evaluation of the solution the problems in the National Traveling Salesman Problem instances offered by the University of Waterloo, which also provides the optimal value of the route of such instances and will allow us to check the quality of our solutions.

On a lower level, the `numpy` package was used for the computations, which enables vectorization of the computations and higher performance in the execution, as well as more expressive and concise code. `pandas` is used for loading the `.tsp` files to memory easily, and `matplotlib` is used to plot the graphical representation. These dependencies are all included in the Anaconda distribution of Python, or can be easily installed using `pip`.

To evaluate the implementation, we will use some instances provided by the aforementioned National Traveling Salesman Problem library. These instances are inspired in real countries and also include the optimal route for most of them, which is a key part of our evaluation. The evaluation strategy consists in running several instances of the problem and study some metrics:

- **Execution time** invested by the technique to find a solution.
- **Quality** of the solution, measured in function of the optimal route: a route that we say is “10% longer that the optimal route” is exactly 1.1 times the length of the optimal one.

The parameters used in the evaluation are the ones found by parametrization of the technique, by using the ones provided in previous works [2] as a starting point. These parameters are:

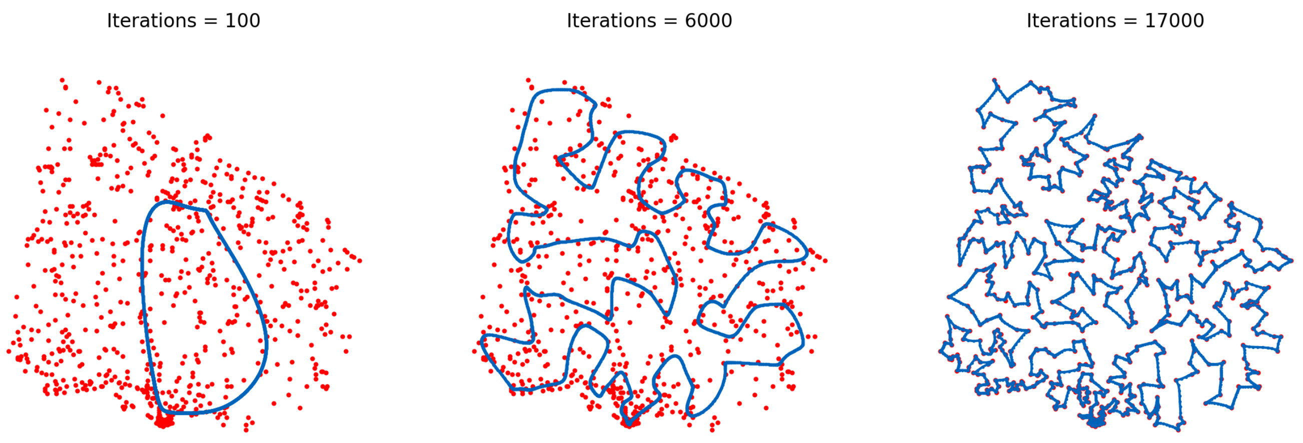
- A population size of 8 times the cities in the problem.
- An initial learning rate of 0.8, with a discount rate of 0.99997.
- An initial neighbourhood of the number of cities, decayed by 0.9997.

These parameters were applied to the following instances:

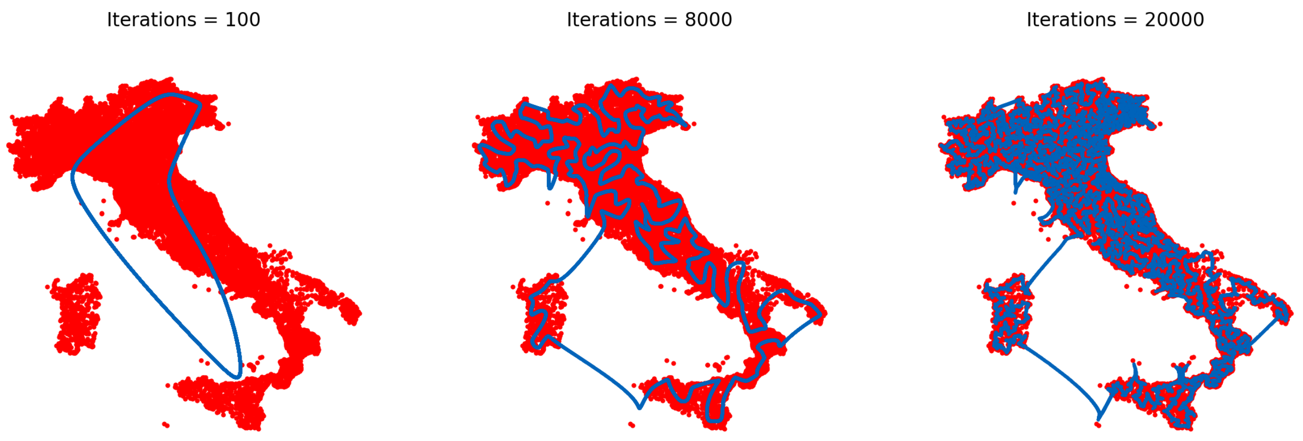
- **Qatar**, containing 194 cities with an optimal tour of 9352.
- **Uruguay**, containing 734 cities with an optimal tour of 79114.
- **Finland**, containing 10639 cities with an optimal tour of 520527.
- **Italy**, containing 16862 cities with an optimal tour of 557315.

The implementation also stops the execution if some of the variables decays under the useful threshold. An uniform way of running the algorithm is tested, although a finer grained parameters can be found for each instance. The following table gathers the evaluation results, with the average result of 5 executions in each of the instances.

Instance	Iterations	Time (s)	Length	Quality
Qatar	14690	14.3	10233.89	9.4%
Uruguay	17351	23.4	85072.35	**7.5%**
Finland	37833	284.0	636580.27	22.3%
Italy	39368	401.1	723212.87	29.7%



The implementation yields great results: we are able to obtain sub-optimal solutions in barely 400 seconds of execution, returning acceptable results overall, with some remarkable cases like Uruguay, where we are able to find a route traversing 734 cities only 7.5% longer than the optimal in less than 25 seconds.



# Final remarks

Although not thoroughly tested, this seems like an interesting application of the technique, which is able to lay some impressing results when applied to some sets of cities distributed more or less uniformly across the dimensions. The code is available in my GitHub and licensed under MIT, so feel free to tweak it and play as much as you wish with it. Finally, if you found have any doubts or inquires, do not hesitate to contact me. Also, I wanted to thank Leonard Kleinans for its help during our Erasmus in Norway, tuning the first version of the code.

All posts | About | Mastodon | GitHub | LinkedIn

1. Kohonen, T. (1998). The self-organizing map. Neurocomputing, 21(1), 1–6. ↩
2. Brocki, L. (2010). Kohonen self-organizing map for the traveling salesperson. In Traveling Salesperson Problem, Recent Advances in Mechatronics (pp. 116–119) ↩