Self-Organizing Maps for solving the Traveling Salesman Problem

Diego Vicente Martín (100317150@alumnos.uc3m.es)

Abstract

Self-organizing maps (also known as Kohonen maps after its creator, Teuvo Kohonen) are an representation technique based on abstracting the data in a grid representation in order to obtain a low-dimensional version of the geometric relations in between. Using this technique, we are able to apply some slight modifications to the grid in order to solve the Traveling Salesman Problem and find sub-optimal solutions for it. In this report, there is an introduction to the concepts of the problem and the techniques used, as well as an evaluation of the Python representation.

1 Summary of the original paper

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).

After this presentation to the technique, the paper suggest several nuances for the implementation that can be useful: using eigenvectors as initialization positions for the model instead of random, using Voronoi regions to speed up the computation of neighborhoods, and how to apply learning vector quantization for discrete classes in the sample vector. Finally, the last remark is the suggestion of using a hexagonal grid when using SOM to create a similarity graph or visualization.

2 Using SOM to solve the Traveling Salesman Problem

The Traveling Salesman Problem (TSP) is an NP-Complete problem consisting of finding the shortest (or, in case of having defined costs, the cheapest) route that traverses all the cities given in a problem exactly once, with or without coming back to the initial point (2).

In this work, the solution proposed uses some slight modifications on the concept of self-organizing maps to use them as a tool to solve the problem. Most of the modifications are inspired by (3) and some of the parametrization techniques are studied in (4).

2.1 Modifying the algorithm

To use the network to solve the TSP, there 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 it and behind. That is, the 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 *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.

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

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.

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. As it is stated in (4), 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.

2.2 Implementation

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.

All the functionalities of the implementation are collected in several files included in the `src` directory:

- `distance.py`: contains the 2-Dimensional (Euclidean) distance functions, as well as other related functions for computing and evaluating distances.
- `io_helper.py`: functions to open `.tsp` files and load them into valid runtime objects.
- `main.py`: includes the general execution control of the self-organizing map.
- `neuron.py`: includes all the network generation, neighborhood computation and route computation functions.
- `plot.py`: some functions to generate graphical representations of different snapshots during and after the execution.

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`.

3 Evaluation

To evaluate the implementation, we will use some instances provided by the 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 (4) 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%

We can see how the time consumption is directly related to the number of cities, since the complexity of finding the closest neuron of a city depends on the population to traverse to find it. It is possible to obtain better results in the large set of cities, but for time constraints it is usually cropped shorter. The results are never 30% longer than the optimal. In the other shorter cases, it is interesting to see how a better quality is obtained in Uruguay even the problem is harder, due to the topography of the country.

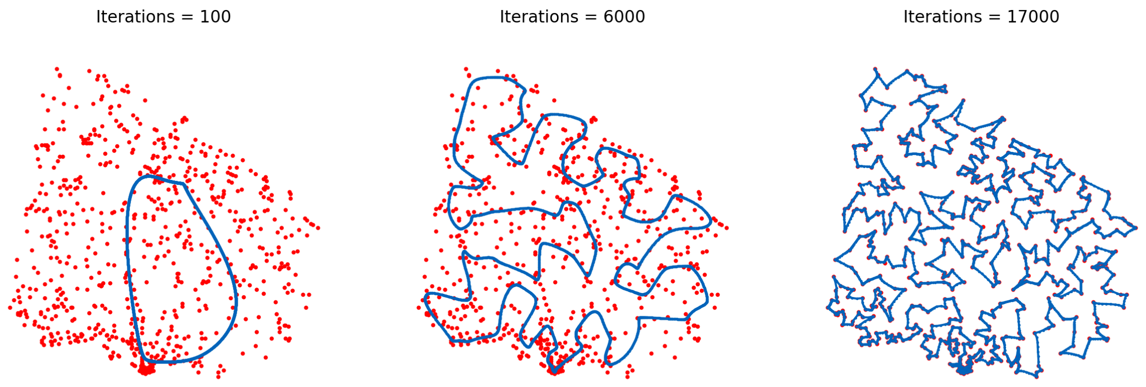

Figure 1: Three different steps of execution of the Uruguay instance.

It is possible to see in the graphical representation of the executions how the technique finds its way: first starting as an elastic, fast changing ring trying to adapt to its shorter form possible, and then starting to fix its position to find better local ways inside of it. For that purpose, a great number of neurons are added to the network and the parameters are decayed, to ensure that first exploration is used and then local exploitation is performed to refine the solution.

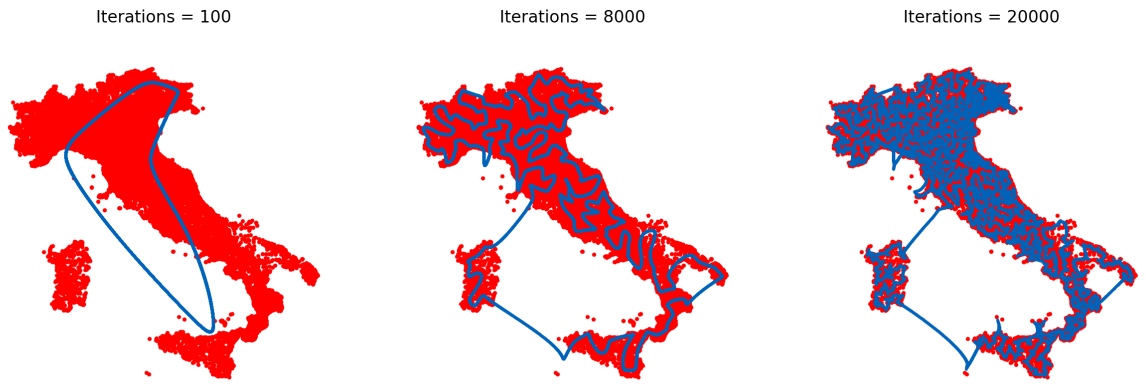

Figure 2: Three different steps of execution of the Italy instance.

The results are overall acceptable, except in extremely demanding settings for the TSP. We can see how in less than half a minute, a suboptimal route crossing more than 700 cities is found.

4 Conclusions

The experiment show a really interesting way to use an organization technique like the self-organizing maps to find sub-optimal solutions for an NP-Complete problem like the traveling salesman problem. By modifying the technique used to find similarity, we are able to create a network that organizes the cities in the one of the shortest routes possible. Even though the technique is sensible to parametrization and may need to run for a great number of iterations in the biggest instances provided, the results are satisfactory and inspiring to find new uses in well-known, established techniques.

References

[1] T. Kohonen, “The self-organizing map,” *Neurocomputing*, vol. 21, no. 1, pp. 1–6, 1998.

[2] K. L. Hoffman, M. Padberg, and G. Rinaldi, “Traveling salesman problem,” in *Encyclopedia of operations research and management science*, pp. 1573–1578, Springer, 2013.

[3] B. Angeniol, G. D. L. C. Vaubois, and J.-Y. L. Texier, “Self-organizing feature maps and the travelling salesman problem,” *Neural Networks*, vol. 1, no. 4, pp. 289–293, 1988.

[4] L. Brocki, “Kohonen self-organizing map for the traveling salesperson,” in *Traveling Salesperson Problem, Recent Advances in Mechatronics*, pp. 116–119, 2010.