IT3708
Subsymbolic Methods in AI

Assignment 3

Applying Self-Organizing Maps to the Traveling Salesman Problem

Kjetil Valle
kjetilva@stud.ntnu.no

Olav Bjørkøy
olavfrih@stud.ntnu.no

NTNU, Trondheim

March 14th, 2010

In this report we describe our results solving the Traveling Salesman Problem (TSP) using Kohonen's Self Organizing Maps (SOMs). We show that this technique is capable of finding relatively paths relatively close to the optimal solution, for a number of different datasets.

1 Introduction

The Traveling Salesman Problem (TSP) is perhaps one of the best known NP-complete problems. There are many approaches to finding the optimal path in such a problem, and in this report we will use Kohonen's Self Organizing Maps (SOMs) [Kohonen, 1998, Downing, 2010b] to calculate approximate and optimal solutions.

The performance of this implementation will be tested against datasets of cities with known optimal solutions, ranging from 29 to thousands of cities. We also show how an approximate solution can be improved during computation by dynamically adding new neurons to uniquely cover each city.

2 Implementation

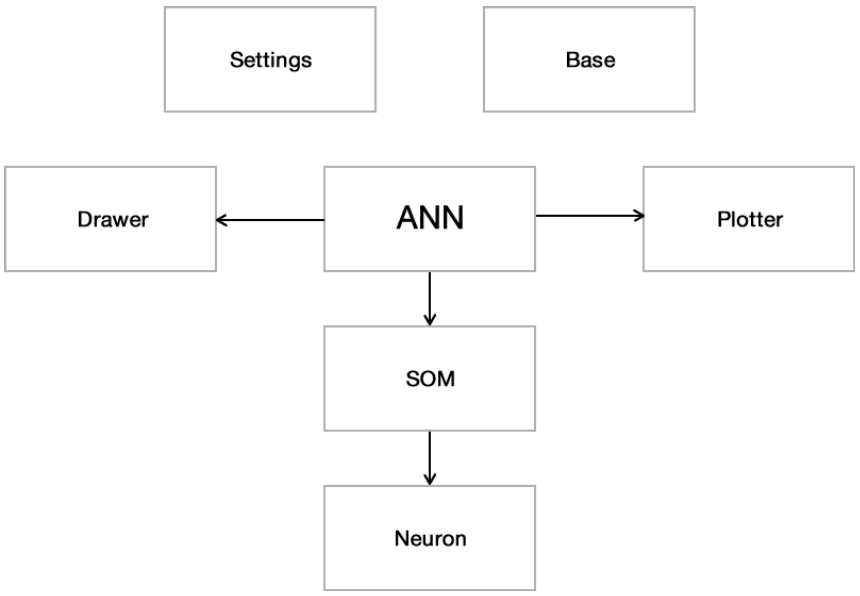

Figure 1: Class diagram for the TSP solver. ANN is the main loop class. Settings is a class available to all other classes. Base is a collection of general utility classes. Drawer and Plotter are used to visualize the calculated solution.

Although the subject of this assignment is quite different from the previous one, we were able to reuse some of our previous implementation. Our Ruby classes for drawing routes, plotting graphs and our general utility classes were all used to solve this assignment. For a detailed description of this part of the implementation, see [Bjørkøy and Valle, 2010].

The code required to implement the SOM was written from scratch. Figure 1 shows the class diagram.

2.1 The Classes

The main class is the `ANN` class. This class bootstraps the application, loads the training data from the input TXT files, and runs the loop which trains the SOM. The `Drawer` and `Plotter` classes are used for drawing and plotting solutions respectively.

The `SOM` class represents the Self-Organizing Map. It contains a list of `Neuron`s, as well as methods for training the neurons using the input data. This class is also responsible for calculating the currently calculated approximate path, i.e. the list of cities corresponding to a walk from neighbor to neighbor in neuron space.

Each instance of the `Neuron` class represent one single neuron. Such an instance holds the neurons weights, corresponding to the coordinates in the TSP domain, and methods for creating random new weights, updating the weights. This class also have a method for moving the neuron close to another neuron (in euclidean space), which is utilized when creating new neurons during computation to deal with clustering cities. (This is described in Section 2.3.)

The `Settings` class holds settings for the application, including task specific settings for a number of datasets.

2.2 Training the neurons

We train each neuron as described in [Downing, 2010b]. Many different combinations of parameters were tested, but results from the tweaking showed that the suggested parameter values from [Budinich, 1996] worked well, and we ended up using something similar to these values.

In the application, the learning rate starts out with a value of 0.8, which is then decreased linearly towards zero in the last epoch of training. The neighborhood radius starts out at around 10% of the total neuron population size, and is decreased linearly to a value of 1.0 after 65% of the training is complete. For the rest of the training epochs, the neighborhood radius stay at 1.0. The number of neurons is initially set slightly higher than the number of cities in the training data. See the Settings class for the exact values for each dataset.

We found that for some datasets, an increased initial neighborhood radius resulted in a better solution. This was especially true for the Djibouti dataset, where we set the initial radius to 20% of the neuron count.

During training we sequentially present the SOM with the normalized coordinates for each city in the current dataset. After training the SOM with all cities, the learning rate and neighborhood radius are updated according to their respective formulas, and the process is repeated. We found that repeating this training loop 50 times (i.e. using 50 epochs), worked well for the datasets used.

When the training of the SOM is complete, we compute the TSP solution given by the trained SOM. This is done by iterating through the cities, and adding each city to the neuron closest to it. We then iterate through the neuron ring, visiting the cities added to each neuron.

If multiple cities are tied to one neuron, we do not know their correct ordering. The next section discuss our proposed solution to this problem.

2.3 Improving the solution

One problem when trying to find the best possible route was that a lot of cities often shared their closest neuron. We could then not tell in which order these cities should be visited when computing the path calculated by the SOM.

A possible solution to this problem would of course be to enumerate all possible orderings of the cities in question. This would have been feasible for small datasets such as "Sahara", but it would not scale well to larger sets where lot of cities have a common closest neuron, as many dataset presents us with multiple clusters of cities.

An alternative solution we used to solve this problem were to dynamically add neurons to cover the clustering cities as needed.

A few times during the training, at epochs specified in the settings, we look for neurons that is the closest one to several cities. When we find such a neuron we add another one close to the original neuron, both in neuron and euclidean spaces. The new neuron is randomly added within a small area around the original neuron. Using this technique, we get more neurons in places where they are needed to distinguish between cities.

We found that adding these extra neurons in the late stages of computation often improved the resulting solution. Figure 2 illustrates the adding of an additional neuron.

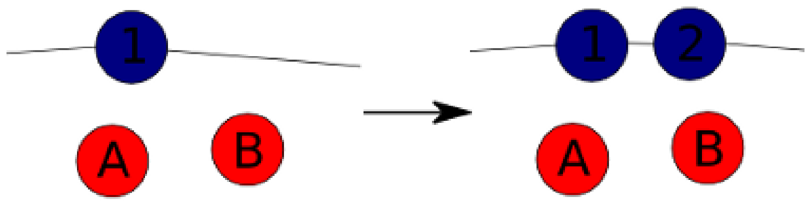

Figure 2: Dynamic addition of a new neuron, randomly placed in a small area around the original neuron, in order to avoid having two cities with a common closest neuron.

3 Results

Table 1 shows a series of test runs with our application, over 4 different datasets.

Dataset	Optimal solution	Average solution	Average error
Western Sahara (29 cities)	27603	27752.02	0.54%
Djibouti (38 cities)	6656	6784.6	1.95%
Qatar (194 cities)	9352	10745.9	14.90%
Luxembourg (980 cities)	11340	12451.0	9.79%

Table 1: Results for the different datasets. Averages is calculated over 5 runs on each dataset.

Each dataset was run 5 times, with the learning rate and neighborhood radius as given in Section 2.2. The collection of city coordinates were presented to the SOM for training 50 times in each run.

As shown in the table, our solution often finds a good, approximate solution to the problem. We found that the smaller the dataset, the higher the probability was for our solution to find the optimal path. For the larger datasets, the average error was a bit higher, but it remained quite stable around the average error given in Table 1.

The approximate routes at different iterations can be seen in Figures 4, 5, 6 and 7. A plot of the progression of the length of the currently calculated path can be seen in Figures 3, 6, 9 and 10.

Table 2 shows results with the Sahara dataset with and without dynamically adding neurons during computation.

Method	**Optimal solution**	**Average solution**	**Average error**
Without extra neurons	27603	28938.94	4.84%
With extra neurons	27603	28069.46	1.70%

Table 2: Results with and without dynamically added extra neurons for the Sahara dataset. Each case was run 5 times.

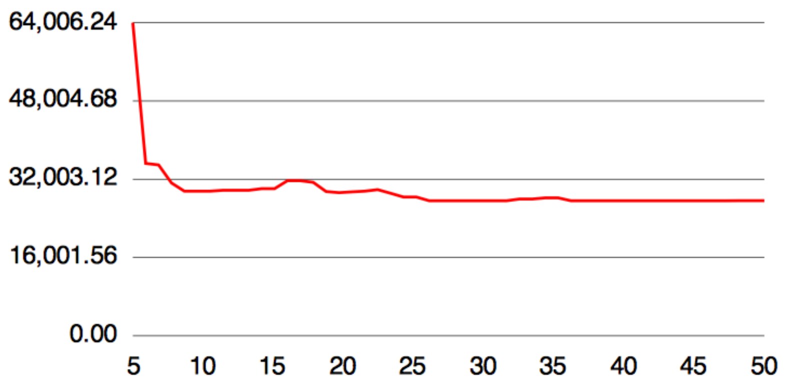

Figure 3: Plot of length of the TSP solution during training for the Sahara dataset. We see that the algorithm quickly converges on a good solution, and uses most of the iterations improving the results slightly.

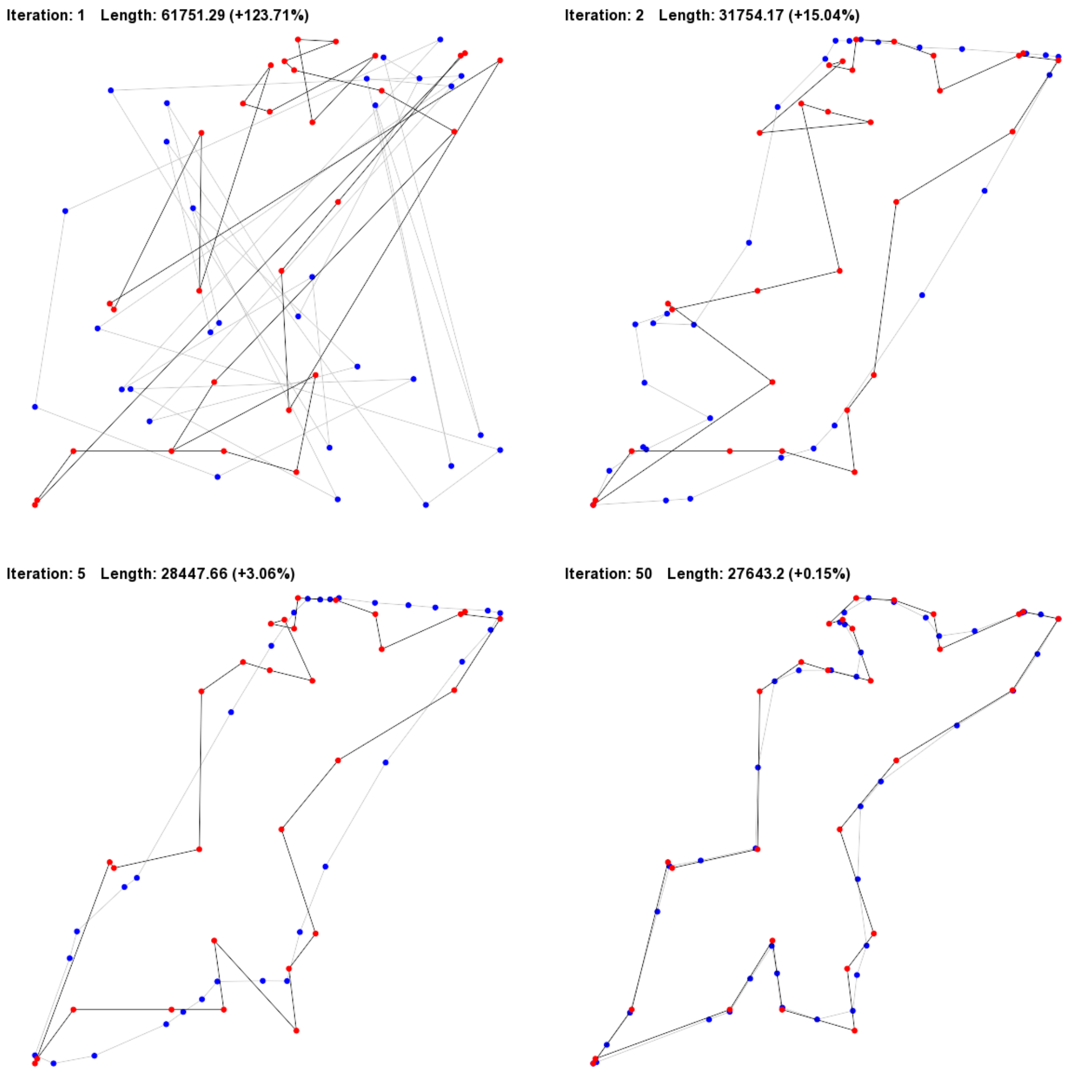

Figure 4: Plots of the SOM and corresponding TSP route after 1, 2, 5 and 50 iterations on the Sahara dataset. Iteration 1 is the random generated SOM, iterations 2 through 50 shows how the SOM reacts to training. The red dots are the input cities, while the blue dots are the current neurons. The black line is the current calculated optimal path, and the grey lines shows the connections between neurons.

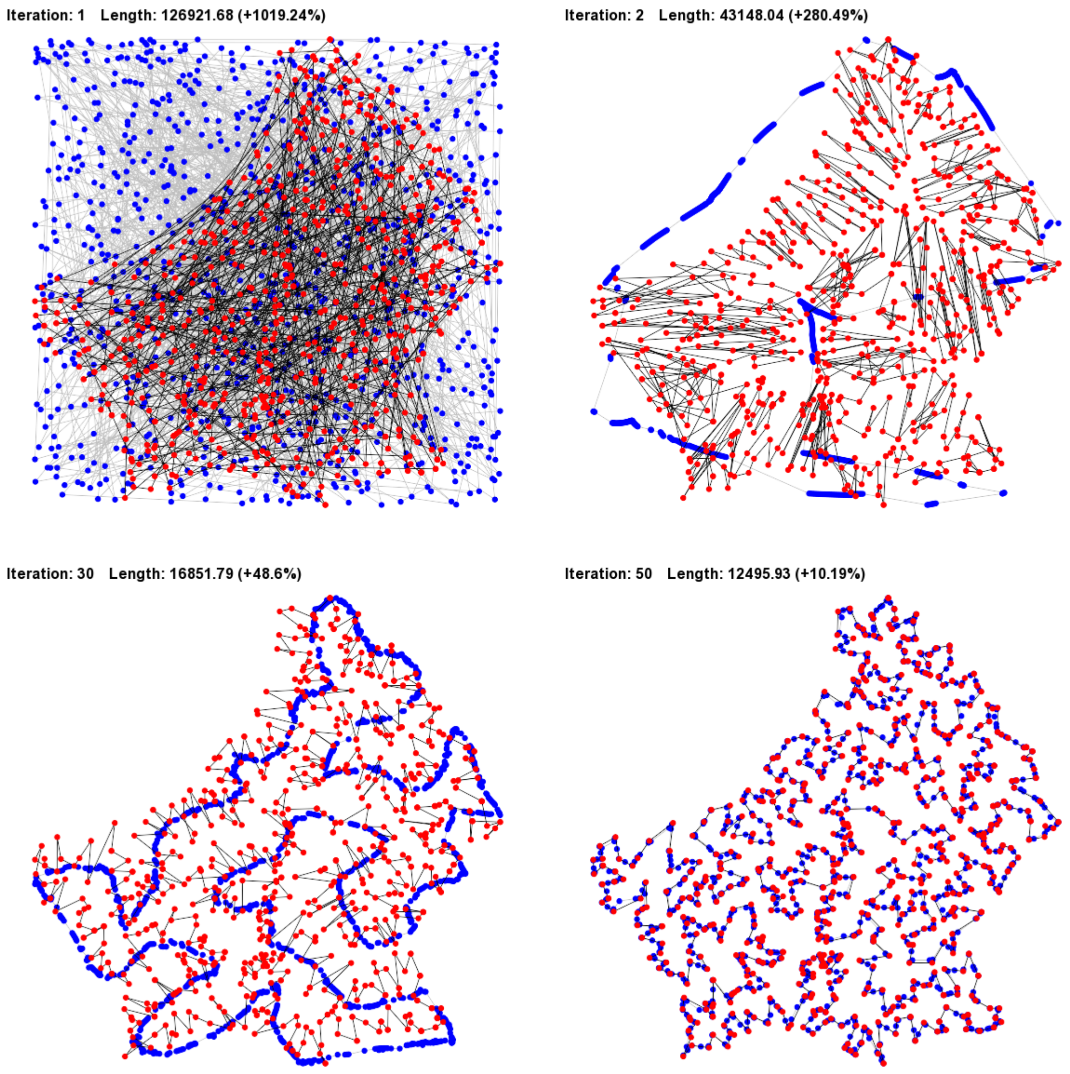

Figure 5: Plots of the SOM and corresponding TSP route after 1, 2, 30 and 50 iterations on the Luxembourg dataset. Extra neurons were added after iteration 20 and 30. The final path is about 10% longer than the optimal solution.

4 Discussion

As the data in Section 3 show, our application reliably finds an approximate solution to the TSP in the datasets we utilized. The algorithm predictably performs best on small datasets, but proved useful even for countries with thousands of cities.

We chose our settings as described in Section 2.2. Our learning rate starts out at 0.8 and decreased linearly towards 0 through the iterations. It is important that the learning rate is high in the beginning so that the initial, randomly placed neurons can approach their closest cities in a high enough degree. Conversely, the learning rate has to be reduced linearly so that the rate of change in neuron placement reflects the actual probable error in the current calculated path.

Our default neighborhood radius starts out at a high value of 10% of the number of neurons. For some datasets, a higher initial radius gave significantly better results, for instance with the Djibouti dataset, which was set at 20% of the initial neuron count.

A high initial radius is needed to make sure the ring of neurons is properly sorted at this early stage in the computation. Towards the end of training, the radius decreases so that a change in one neuron's weight does not negatively influence other neurons that might already be weighted correctly. Through experimentation and tweaking, we found that keeping the radius at 1 after 65% of the training iterations gave the best results.

Some datasets were more troublesome than others. Sets with many clusters of large number of cities were the biggest challenge for our solution, as seen in Figure 7. Our technique for dealing with these clusters, by dynamically adding more neurons during computation where they are needed, seems to mitigate this problem quite well, as seen in Table 2.

In our implementation, the extra neurons are added at predetermined iterations, namely iterations 20 and 30. An improvement to this technique would be to programmatically decide when extra neurons should be added, for instance by adding neurons at given percentages of the total number of iterations.

Our implementation of the algorithm is not general or meant to be used for other problems, as this was not a part of the exercise. However, by sacrificing some generality, we achieved a good solution through a small codebase, while still retaining a good deal of modularity. By opting for the same implementation language in this exercise as the other assignments, we were able to save us some time by reusing our existing visualization and utility classes.

References

[Bjørkøy and Valle, 2010] Bjørkøy, O. F. and Valle, K. (2010). Assignment 2 - solving traveling salesman problems with evolutionary algorithms. [Assignement 2 delivery in IT3708 - Subsymbolic Methods in AI, Spring 2010].

[Budinich, 1996] Budinich, M. (1996). A self-organizing neural network for the traveling salesman problem that is competitive with simulated annealing. *Neural Computation*, 8(2):416–424.

[Downing, 2010a] Downing, K. L. (2010a). Homework module: Applying self-organizing maps to the traveling salesman problem(tsp). [Available from `http://www.idi.ntnu.no/emner/it3708/assignments/modules/ann-som-tsp.pdf`, Accessed 2010-03-04 10:32].

[Downing, 2010b] Downing, K. L. (2010b). Unsupervised learning in neural networks. [Available from `http://www.idi.ntnu.no/emner/it3708/lectures/learn-unsup.pdf`, Accessed 2010-03-02 14:17].

[Kohonen, 1998] Kohonen, T. (1998). The self-organizing map. *Neurocomputing*, 21(1-3):1–6.

Appendix

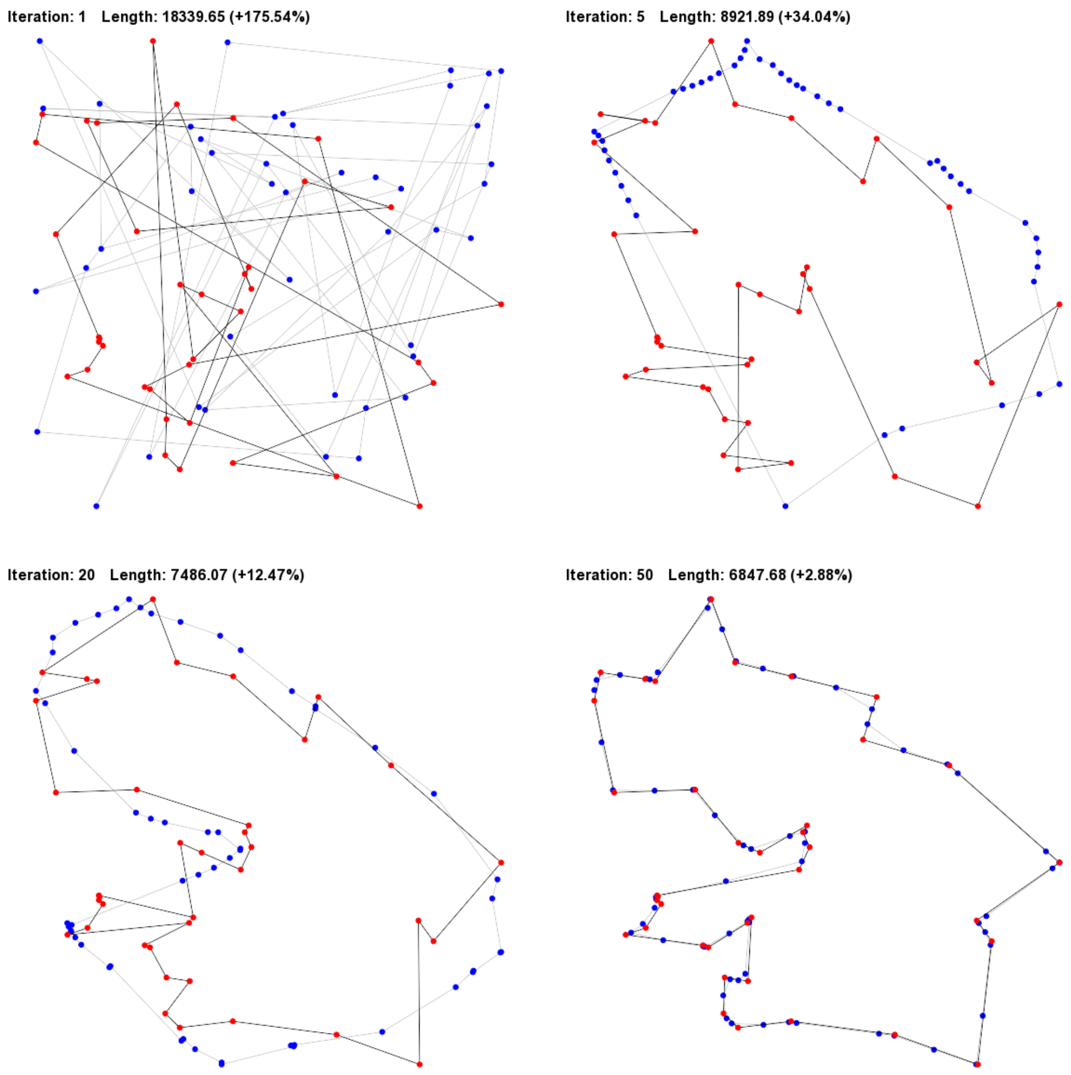

Figure 6: Plots of the SOM and corresponding TSP route after 1, 5, 20 and 50 iterations on the Djibouti dataset. Extra neurons were added after iteration 20 and 30. The final path is about 3% longer than the optimal solution.

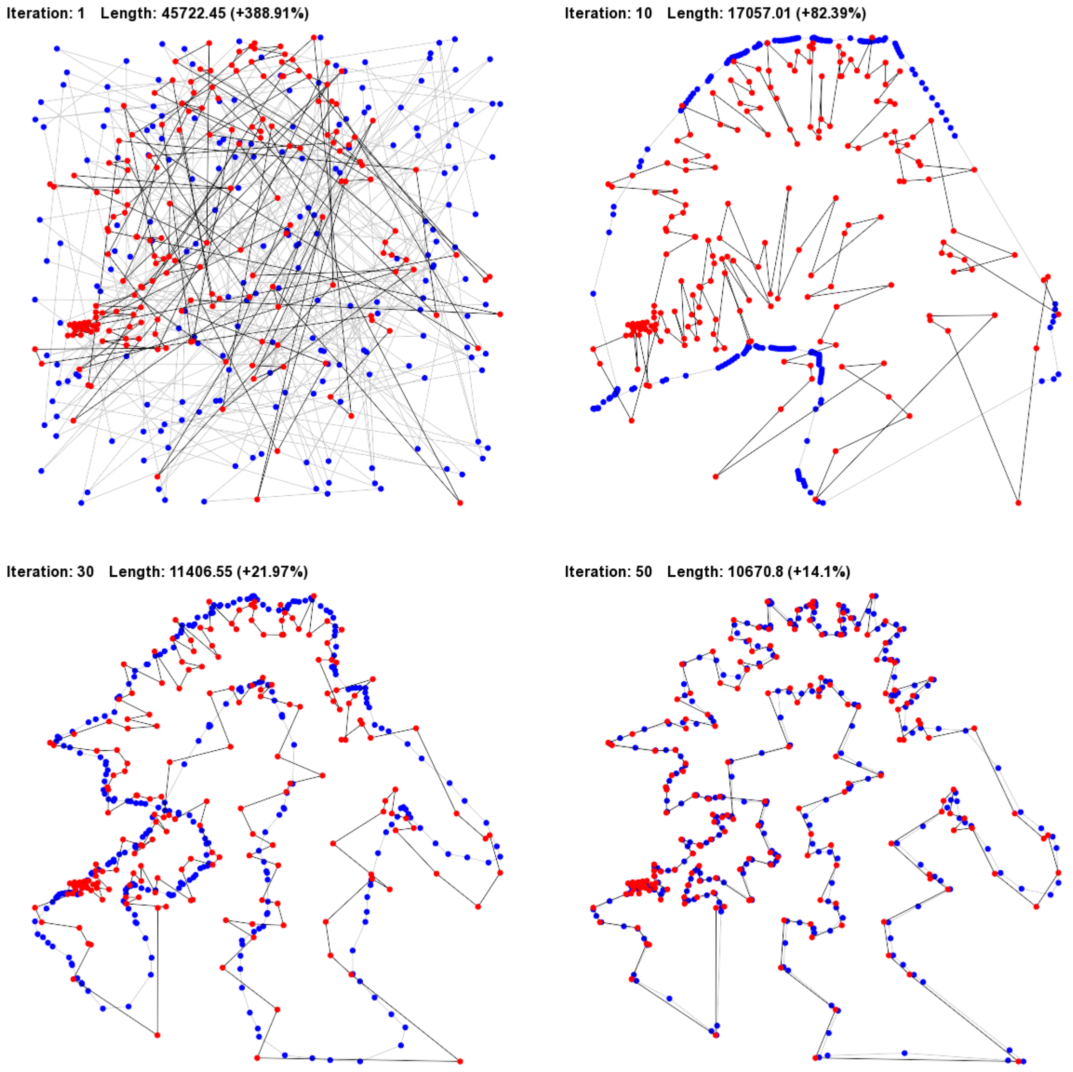

Figure 7: Plots of the SOM and corresponding TSP route after 1, 10, 30 and 50 iterations on the Qatar dataset. Extra neurons were added after iteration 20 and 30. The final path is about 14% longer than the optimal solution.

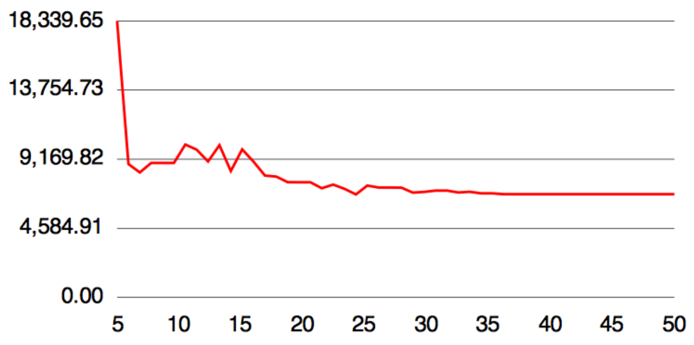

Figure 8: Plot of length of the TSP solution during training for the Djibouti dataset. We see that the algorithm quickly converges on a good solution, and uses most of the iterations improving the results slightly.

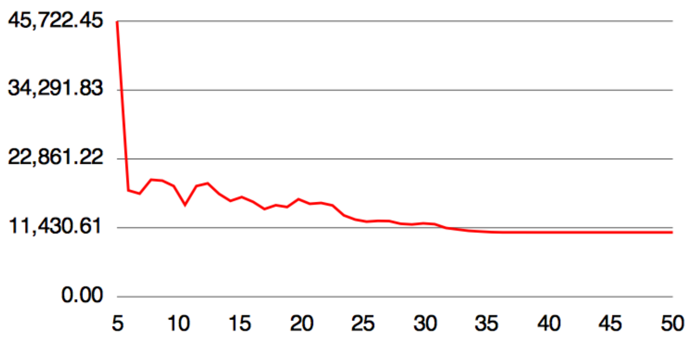

Figure 9: Plot of length of the TSP solution during training for the Qatar dataset. We see that the algorithm quickly converges on a good solution, and uses most of the iterations improving the results slightly.

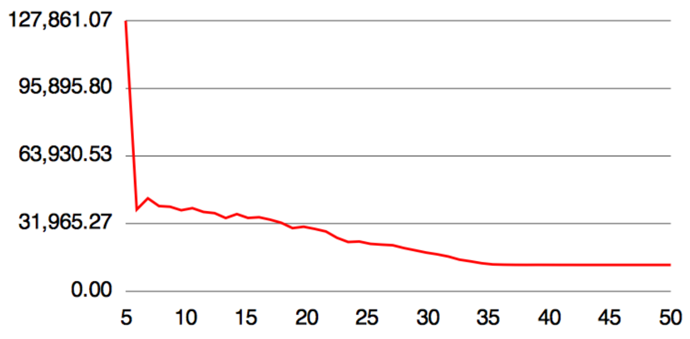

Figure 10: Plot of length of the TSP solution during training for the Luxembourg dataset. We see that the algorithm quickly converges on a good solution, and uses most of the iterations improving the results slightly.