Project 3.5 in Artificial Intelligence Programming - IT3105

Knut Andreas Hasund

February 1, 2014

Abstract

Applying Self-Organizing Maps to the Traveling Salesman Problem.

The Traveling Salesman Problem

The traveling salesman problem or **TSP** for short is defined as such: Given a set of cities where the distance between each pair of cities is known. Find the shortest possible path to take that visits each city only once and also ends up in the starting city?
In this project I have looked at how a good (not perfect) solution can be found using Self-Organizing Maps.

Implementation

I have decided to implement my system with a rather simple architecture. It is divided into three classes, `PointSet`, `Neurons` and `TSPSolver`.

PointSet

```
class PointSet {
public:
    PointSet(const std::string &filename);
    std::shared_ptr<std::vector<int> > getRandomOrder();
    std::pair<double,double> getWeights(int pointIndex);
    int getDimension();
    void print();
};
```

This class contains each city in the **TSP** problem. It is constructed using the name of the text file that contains the city data. All parsing and storing of data is then done automaticly. All city coordinates is mapped to the normalized domain $[0,1] \times [0,1]$ which should make the algorithm less exposed to issues related to badly scaled parameters. An option for generating a randomly ordered index list is also avaiable.

Neurons

```
class Neurons {
public:
    Neurons();
    Neurons(int numberOfNeurons);
    std::pair<double, double> getWeight(int neuron);
    std::shared_ptr<std::vector<std::pair<double,double> > > getWeights();
    int getDimension();
    void setWeight(int neuron, double x, double y);
    void setWeight(int neuron, std::pair<double,double> weight);
    void print();
};
```

Contains the connected ring of neurons. Each neuron on the ring is initialized by giving it a random position on the $[0,1] \times [0,1]$ domain. Allthough faster convergence probably can be found by deciding the initial placement by exploiting the given data set or trends among them, a method of random or at least pseudo random initial placement should be advantagous in cases where several cases of the same experiment is ran. This due to increased chances that there will be found different(even better) solutions when solving the problem several times with random initialization. Besides initialization this class is mostly a container for the neurons used by the main algorithm.

TSPSolver

```
class TSPSolver {
public:
    TSPSolver(const std::string filename, int k, int maxit, double alpha_0,
                        double omega_0, double alpha_tau,  double omega_tau);

    void runAlgorithm();
    std::vector<int>  findPath();
    double findPathLength(std::vector<int> path);

    void writePathToFile(int  iteration , std::vector<int> path);
    void writeStatusToFile(int  iteration);
    void  printStatus();
};
```

This is the main work horse of the system, and almost all the algorithmic work towards the solution is done here. Except for the main algorithm, the class contains utility for writing log files for plotting, and printing directly to the system console (not recommended for large systems).

The implementation of the self-organzing maps algorithm has been done using two set of vectors, each containing N cities and M neurons respectively. While the cities are topologicaly connected via their coordinates in the $[0,1] \times [0,1]$ domain, the neurons are connected via its neighbours in the vector. This means that a neurons closest topological neighbour is the two neighbours closest to it in the vector (the vectors endpoints is connected in this regard). The neurons are not shuffeled around, thus remaining connected through the whole runtime of the algorithm. For each iteration each of the cities are presented in a random order to the neuron network. The neuron with closest coordinates to the city is selected and marked so that it is unable be chosen as the closest node for several cities in the same iteration. This allows more nodes to have induvidual movement each iteration. The selected neuron and its topological neighbours then have their coordinates (weights) updated according to the equation.

$$w_i^{new} = w_i^{old} + \alpha e^{(-D^2/\omega^2)} \left(W_i - w_j^{old} \right) \tag{1}$$

where D is the absolute value of the index distance between the selected neuron and a given neighbour, w_j is the jth neurons weight and W_i is the ith city's weight. α is the learning rate and ω is the variance in the neighbourhood function.

As one can see the learning for each neuron is dependant on the learning rate, the proximity to the city's closest neuron and its distance from the city itself. The values of both α and ω are changed during the algorithm as according to the heuristics proposed by Kohonen[1],

$$\alpha_k = \alpha_0 e^{\left(-\frac{k}{\tau_\alpha}\right)} \tag{2}$$

$$\omega_k = \omega_0 e^{\left(-\frac{k}{\tau_\omega}\right)} \tag{3}$$

where k is the iteration number. Since the neighbourhood function becomes low as the algorithm progresses the total number of neighbours a neuron can influence is capped at a

[1]Kohonen, Teuvo, Self-Organizing Maps, 3rd ed. 2001, Springer

percentage. I addition to this as the algorithm propagates out from the original neuron along the neighbours, it will stop and continue with the next city if the relative change to the weight of a neighbour is too low compared to the original neuron's change. This in hope of reducing an unneeded number of calculations. To see how the neighbourhood function changes with varying input, see figure 1.

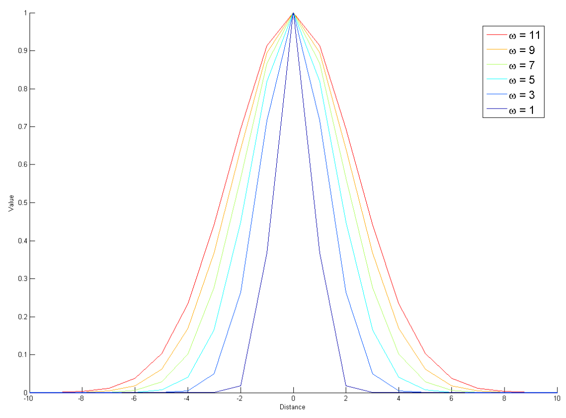


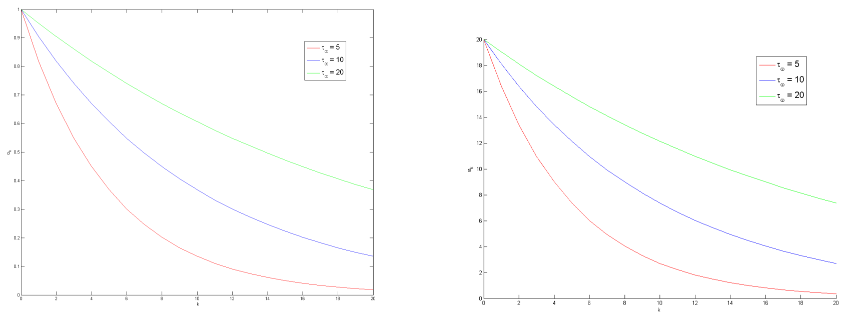
Figure 1: The neighbourhood proximity weight $e^{(-D^2/\omega^2)}$ as a function of D with different values of ω

.

The class also have the ability to calculate the current found path at any point in the algorithm.

Tuning

I found this algorithm fairly hard to tune properly. Looking at the values for the neighbourhood function and the learning rate, I concluded that setting parameters so they where just above zero in the algorithms last iteration was improving results. This is because we want the neurons to settle towards the end of the algorithms designated runtime. It was hard to tell how to calibrate the curve of the reduction, as I got pretty similar results, though it seems a smooth decent is better. When it comes to the learning rate, if the curve becomes too steep, the algorithm will stop making progress as the neurons no longer are able to move much from their established places. Thus it will converge to a local minima (which might be suboptimal). On the other hand will a too slow curve make the algorithm unstable, with neurons jumping all over the place due to the randomness of city iteration each timestep. It is much the same for the neighbourhood function, just with larger groups of neurons at once. If neighbouring nodes pull too much on eachother there will be no local convergence, but if they pull to little it is impossible to get a good smooth spread of the neurons (see figure (**??**) for the behaviour of α and ω for different τ values).



(a) α_k **(b)** ω_k

Figure 2: Curves of parameters as a function of k with different τ values

Its also necessary to establish the two initial values of α and ω. In regards to α it makes no sense to have a learning rate that is over 1, since that would make the neurons over shoot their target. For ω on the other hand, it becomes a bit more tricky. Ideally we would want quite large neighbourhoods at the start. I have had good results with values between a fifth and a half of the number of neurons M as initial value.

Results

Below are the results obtained solving the three maps: Djibouti(38 cities), Qatar(194 cities), and Oman(1979 cities). I changed this since I noticed Dijbouti's map was changed since the assignment was given originaly. I hope it is ok that i only present the solution for Western-Sahara. Everything is plotted using `MATLAB`.

Western-Sahara - 29 cities

Solved using $\alpha_0 = 1$, $\omega_0 = 14$, $\tau_\alpha = 10$, $+tau_\omega = 5$, maximum iterations equal 20.

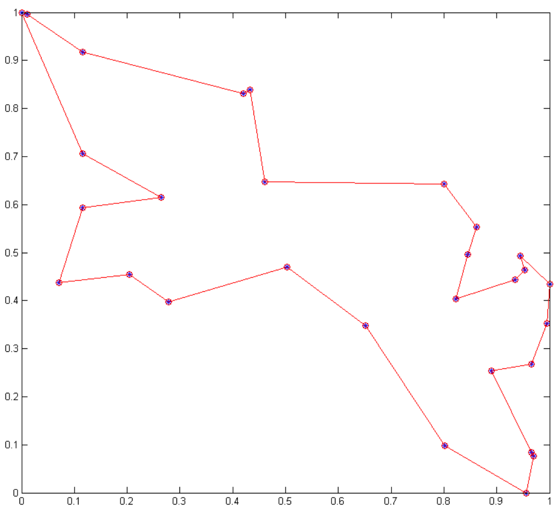
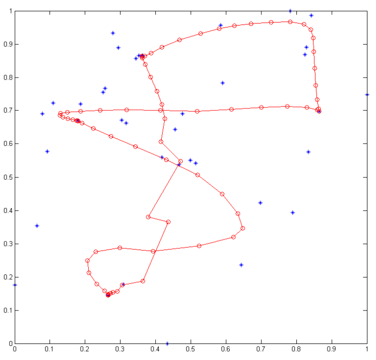


Figure 3: Last step, $D_{path} = 3.984$

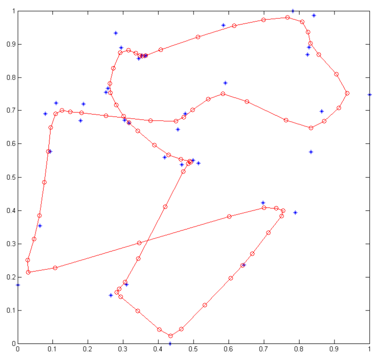
.

Djibouti - 38 cities

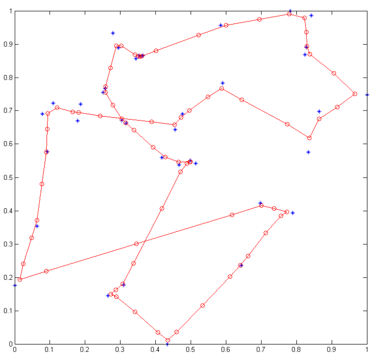
Solved using $\alpha_0 = 1$, $\omega_0 = 15$, $\tau_\alpha = 10$, $+tau_\omega = 5$, maximum iterations equal 20.



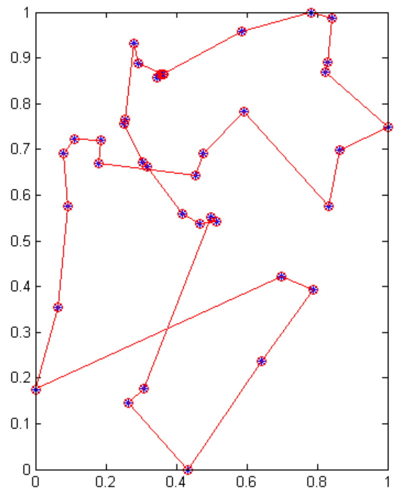
(a) Step 2, $D_{path} = 5.599$)



(b) Step 7, $D_{path} = 4.543$



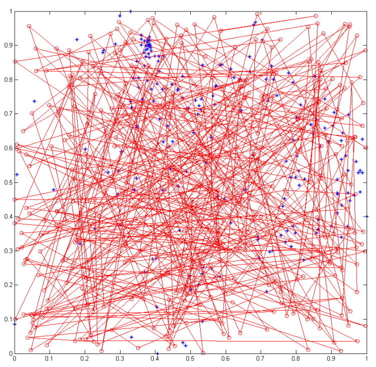
(a) Step 19, $D_{path} = 4.243$



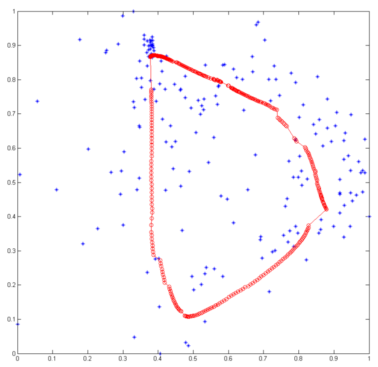
(b) Last step, $D_{path} = 4.242$

Qatar - 129 cities

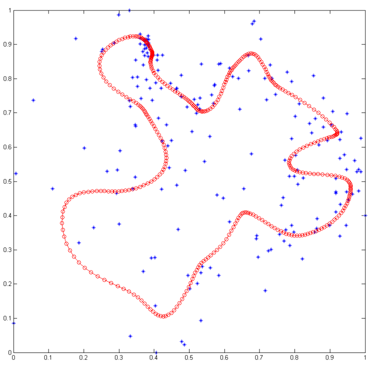
Solved using $\alpha_0 = 1$, $\omega_0 = 100$, $\tau_\alpha = 10$, $+tau_\omega = 5$, maximum iterations equal 40.



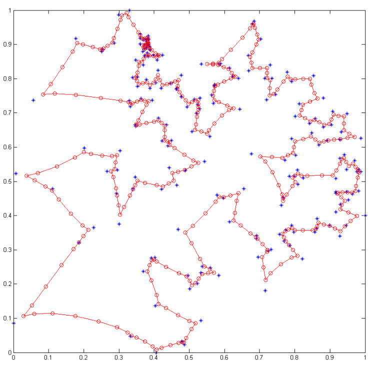
(a) Step 0, $D_{path} = (\textit{not calculated})$



(b) Step 4, $D_{path} = 15.456$



(a) Step 12, $D_{path} = 10.684$



(b) Step 28, $D_{path} = 9.174$

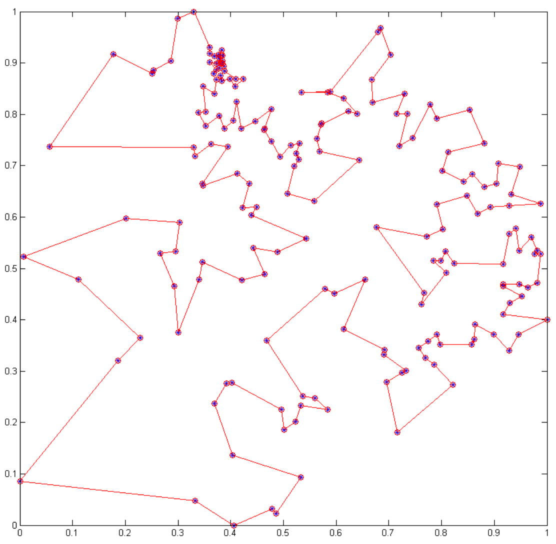
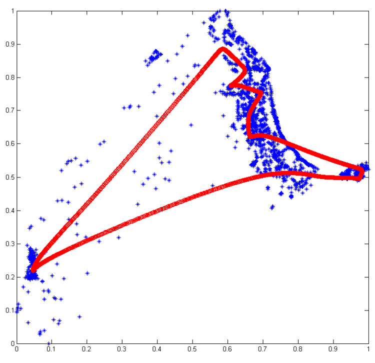


Figure 8: Last step, $D_{path} = 9.172$

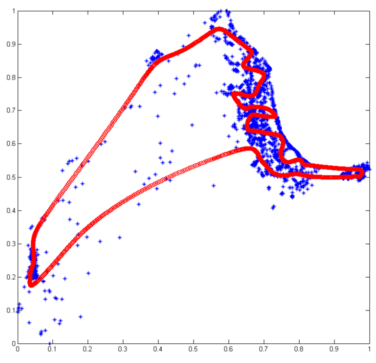
.

Oman - 1979 cities

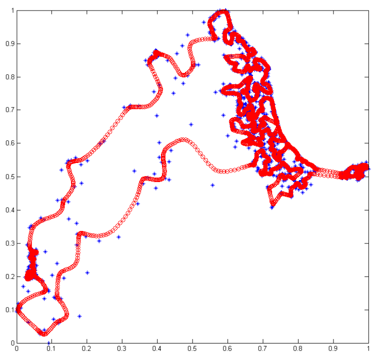
Solved using $\alpha_0 = 1$, $\omega_0 = 300$, $\tau_\alpha = 10$, $+tau_\omega = 5$, maximum iterations equal 40.



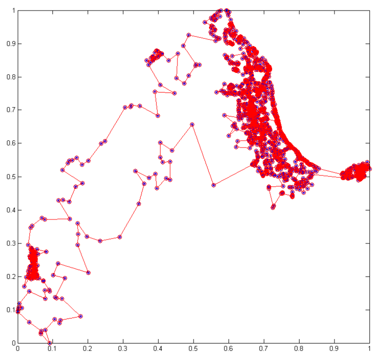
(a) Step 0, $D_{path} = 41.711$



(b) Step 8, $D_{path} = 23.587$



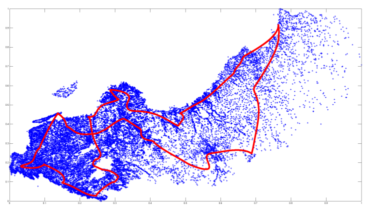
(a) Step 20, $D_{path} = 11.876$



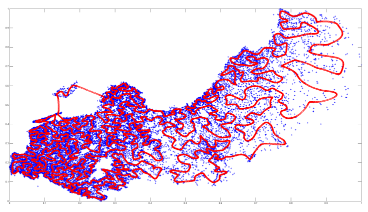
(b) Last Step, $D_{path} = 11.312$

Bonus Content: Sweden - 24,978 cities

Solved using $\alpha_0 = 1$, $\omega_0 = 1000$, $\tau_\alpha = 20$, $+tau_\omega = 5$, maximum iterations equal 50.



(a) Step 2, $D_{path} = 530.6$



(b) Step 18, $D_{path} = 127.3$

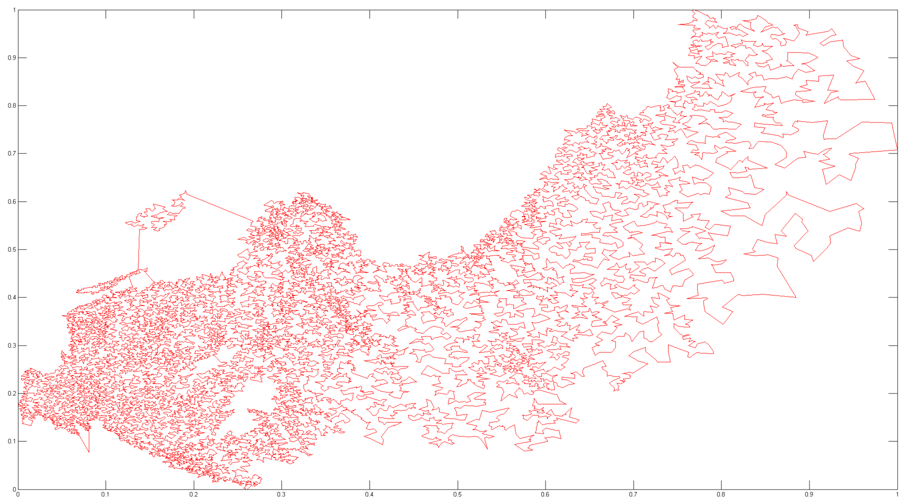


Figure 12: Last step, $D_{path} = 69.7$

.