IT3105 project3 report

Fredrik Ingebrigtsen

November 2016

1 System Description

Here is a broad overview of the SOM algorithm used. Following is a closer look at our implementation.

1. Initialize nodes.
2. For each city
   (a) Find best matching unit (bmu)
   (b) Update node weights in neighborhood of bmu.
3. Repeat from 2.

The nodes was to begin with initialized at random, but laying them out in a circle produced much better results. Selecting a good initial solution is a common problem, and I'm sure there better alternatives than just a circle as well (e.g using a small random selection of the data as a starting point), but our approach seemed sufficient.

We call one traversal of all the input cities an *epoch*. In each epoch the learning rate and radius are first calculated (they remain constant throughout the epoch). Then each city is traversed where the bmu is found, the neighborhood indexes are calculated and the corresponding nodes' weights are updated using the following equation:

$$W(n) = W(n) + \theta(n, bmu) * \alpha * (city - n)$$

α is the learning rate, θ an influence scalar based on distance from the bmu. n and city are 2-d weight vectors. The user controls when and how many epochs should be calculated and when to calculate a TSP solution. The TSP solution is calculated by again traversing each city, associating it with its bmu. The TSP tour is then each city following the order of the bmus. In the case that two or more cities have the same bmu, the order is chosen at random.

Our application is written in C++14 and Qt (gui). The input is read supplied in .txt files on the TSPLIB format. First the cities are normalized onto a 1 by 1 grid. Then because the cities were often listed in geographical order, and we wanted to traverse them randomly, we shuffled the containing vector. We used a std::vector (std::vector<std::pair<double, double>>), as opposed to a map as a container because the main operation performed is traversal, and not much look up (perhaps a raw static array would be best since we are not changing the size after initialization. This was not tested). Using float instead of double gave a very slight performance gain, so we went with the extra precision.

The system has the unfortunate feature of non deterministically crashing. We suspect there may be some memory leakage.

2 Parameter Tuning

The performance of the system depended heavily on the parameters chosen.

It was quite critical that the learning rate was relatively small and decreasing, but not zero. The exponential decay we found was the best fit although linear worked with the exception that it stops learning when it hits zero. A static low learning rate may work in conjunction with different radius decay, but that was not part of the project.

As for the radius, we initially thought that static decay was not suitable because the network works best when it first adjusts to the overall outline of the map, and later more detailed, but the static decay sometimes gave better results than linear. This is perhaps because as long as there is enough nodes to distribute (and small enough radius), then it doesn't matter that some of the nodes remain in starting position when the starting position is a circle, meaning that they have an okay tour to begin with. The radius also was a big decider of the running time, since the epochs with large radius has more node weights to update and take proportionally more time to calculate. We found that the more cities there were, the smaller of a fraction the starting radius could be.

One important consideration for tuning the parameters is how long you want the algorithm to run, or how many epochs. The more epochs you run may give a better results but interestingly we found this gain to be small, and even that the larger data sets did not require significantly more epochs to get satisfying results. Around 10-20 epochs looked enough for both Djibouti (38 cities), as well as for Burma (33704). It is important though that decay functions decay according to the amount of epochs, so that they do not decrease to fast nor slow.

The number of nodes created is another parameter that needed tuning, although not much. As long as there were more nodes than cities the system performed well. We settled on a factor of 1.25 the number of cities.

2.1 Optimizations

The algorithm spends most of its time calculating the bmu for the cities, hence a target for optimization. The bmu is the node with the least distance (euclidean) to the city. To find it we have to traverse all the nodes which is O(num of nodes). To speed this up we parallelized it using OpenMP (straightforward when finding

	Static	Linear	Exponential
Western-Sahara	30 784	28 371	27 872
Djibouti	7155	7523	6951
Qatar	12 937	12 228	11 173
Uruguay	112 377	120 941	87 671

Table 1: Total TSP distance with different decay.

minimal element/index of a list), and since square root is a monotonic function it suffices to compare the norm.

The influence function, which scales the learning of neighborhood nodes exponentially with the distance from bmu, was another expensive function eating a lot of time, that only improved accuracy slightly. In our experiments having a gaussian influence function verses a static gave roughly 1-2 % better total distance (and roughly 20-30 % slower system) (a linear function was not tried out). We left it as a user-selectable option.

3 Results

$$numberofneurons = C * 1.25$$

$$initialneighborhoodsize = C/n$$

$$static \quad R_t = 0$$

$$linear \quad R_t = t * R_0/m$$

$$exponential \quad R_t = \mathrm{e}^{4t/m}$$

$$static \quad \alpha_t = 0$$

$$linear \quad \alpha_t = t * R_0/2m$$

$$exponential \quad \alpha_t = \mathrm{e}^{t/4m}$$

Where α is the learning rate, R radius, t is timestep (epoch number), C is the number of cities, while $2 < n < 16$ and $10 < m < 20$ are some constants.

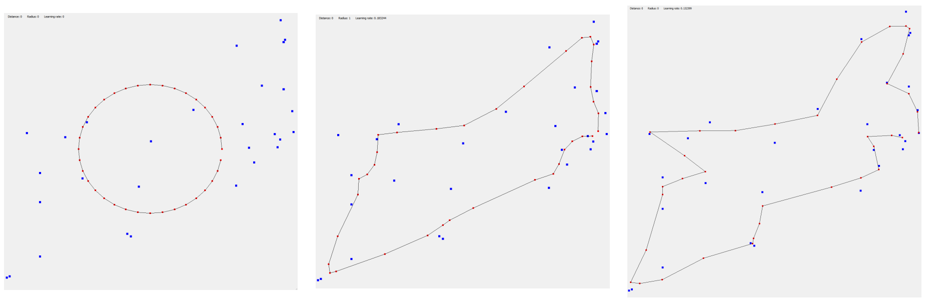

Figure 1: Western-Sahara (29). Beginning, middle and end.

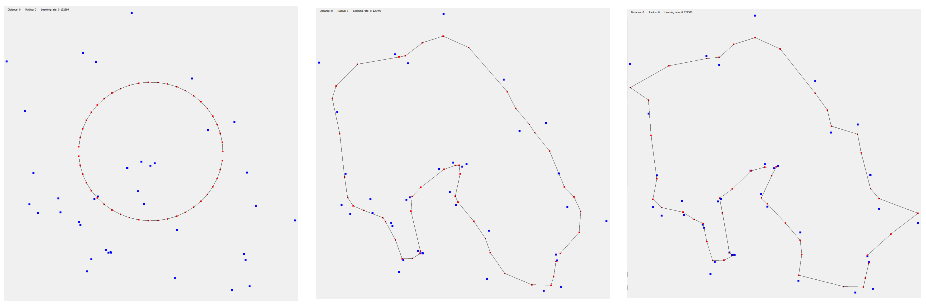

Figure 2: Djibouti (38). Beginning, middle and end.

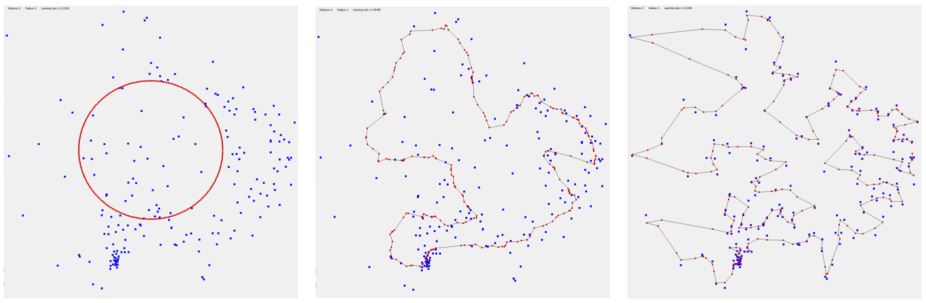

Figure 3: Qatar (194). Beginning, middle and end.

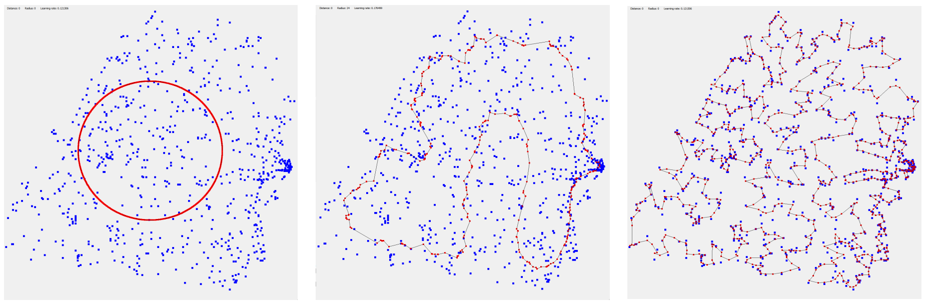

Figure 4: Uruguay (734). Beginning, middle and end.

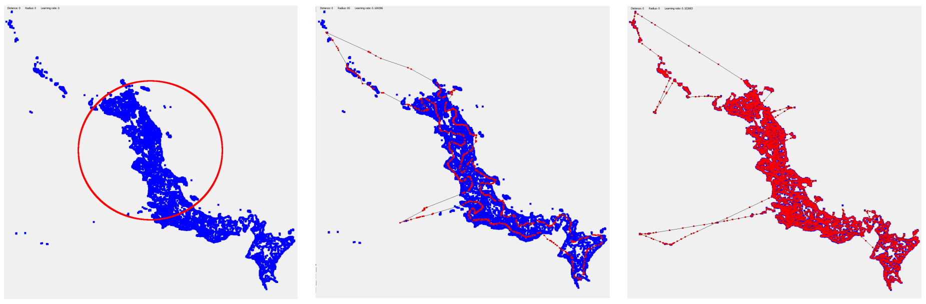

Figure 5: Japan (9847). Beginning, middle and end.