Exercise 3 - SOM MetroMap Implementation

Julius Piso, simmac

January 31, 2021

Abstract

In this work we present our implementation of a MetroMap visualization framework for self-organizing maps. Our solver is based on a branch-and-bound-like algorithm. We explore how varying some parameter settings influences the final appearance of the metro map visualization. Furthermore we compare our results with the ones obtained using the *SOMToolbox* [1]. The complete source code is available on a `https://github.com/yozoon/SOM-MetroMap`.

1 Implementation

1.1 Grid Snapping / Octilinearity

For enforcing octilinearity constraints, we used the description in the original SOM MetroMap visualization paper [1] as orientation and extended it with additional constraints designed to minimize (sharp) corners.
In order to adjust the resolution, and therefore the accuracy and complexity, of the metro map visualization, we introduced an additional grid on top of the unit grid of the SOM. This allows us for example to make large SOMs with many components more easily understandable by reducing the resolution and therefore complexity of the visualization. In order to work on this metro grid, we scale the coordinates of the region centers to the metro grid size for the grid snapping process.
For each component, we start by generating a neighborhood for the first region center which consists of points on our metro grid which are near the region center. We do this by first snapping to the nearest point (rounding to the nearest integer coordinates) and then walking concentric squares around this point on the coordinate grid until we visited a configurable amount of points. For each point, we calculate the (Euclidean) distance to the region center, sort the list of points by their distance to the region center and return the first l elements.
After that, we iterate through each neighbor, starting from the one with the smallest distance and call our recursive `snap_lines()` function, which works

[1] `http://www.ifs.tuwien.ac.at/dm/somtoolbox/index.html`

similar a branch-and-bound algorithm with a non-estimated and one-sided bound, described in Algorithm 1.

Algorithm 1: Snapping algorithm

```
Data: components where each component consists of a list of region
      centers.
Result: solutions where each solution is a point of coordinates
        representing the stops in the metro line for the respective
        component
Function snap_line(unsnapped_points, snapped_points, weight,
 upper_bound):
    if unsnapped_points.size = 0 then
        return weight, snapped_points;
    point ⟵ unsnapped_points.pop(0);
    neighbors ⟵ gen_feasible_neighbors(snapped_points, point);
    best_solution ⟵ Ø;
    for neighbor ∈ neighbors do
        if neighbor.weight + weight > upper_bound then
            return ∞, Ø;
        weight, solution ⟵ snap_line(unsnapped_points,
          snapped_points + neighbor, weight, neighbor.weight,
          upper_bound);
        if weight < upper_bound then
            upper_bound ⟵ weight;
            best_solution ⟵ solution
    return upper_bound, solution;
begin
    for component ∈ components do
        start ⟵ component[0];
        neighbors ⟵ generate_neighbors(start);
        upper_bound ⟵ ∞;
        best_solution ⟵ [];
        for neighbor ∈ neighbors do
            weight, solution ⟵ snap_line(component, [],
              neighbor.distance, upper_bound);
            if weight < upper_bound then
                upper_bound ⟵ weight;
                best_solution ⟵ solution;
        add best_solution to solutions;
```

The `gen_feasible_neighbors(snapped, point, k)` function generates k points with weights in the neighborhood of the point which are reachable from the last snapped point (so in a gridline or diagonal line with the last snapped point). The weight of a point consists of the (Euclidean) distance to the point

and an additional (optional) penalty which is dependent on the angle to the last snapped line that a new line to this point would create. This allows us to reduce the number of sharp corners in the metro map at the cost of geographical accuracy in order to make the generated maps less cluttered and making the general direction of the component gradient more clearly recognizable.
The run-time of our algorithm is in $O(l * k^{b-1} * c)$, where l is the number of neighbors of the first region center, k is the number of neighbors generated for each region center after the first, b is the number of bins per component and c is the number of components. The bounding should reduce the practical run-time considerably. Because the run-time scales exponentially with the neighborhood-size, we implemented a little optimization in our neighborhood-generation: We generate more than b neighbors, sort the neighbors by their weight and return the first b neighbors. This increases the local neighborhood search space without increasing the search space of the underlying problem and should somewhat increase the quality of our results without impacting the run-time too much.

1.2 Post-Processing

Surprisingly, one of the most difficult parts of this exercise was the implementation of the post-processing algorithm. Our post-processing routines shift stations on overlapping metro lines, such that each individual line is visible on the final output. Additionally we collect a list of crossover stations, which can then be used to draw special station shapes on the final visualization. We implemented this functionality by following the component lines on the metro grid and storing for each point and orientation (up/down, left/right and the two diagonals) of this particular edge how many other lines share at least one grid point with the same orientation. We then assign this value to the stations that enclose the edge, as well as all edges that lie on the same straight line. Finally we shift all stations orthogonally to the orientation of the affected adjacent edges by the amount required to separate overlapping lines.

2 Parameter Tuning

2.1 Corner-Penalty

For the first experiment we wanted to figure out how changing the corner penalty values influences the final result.

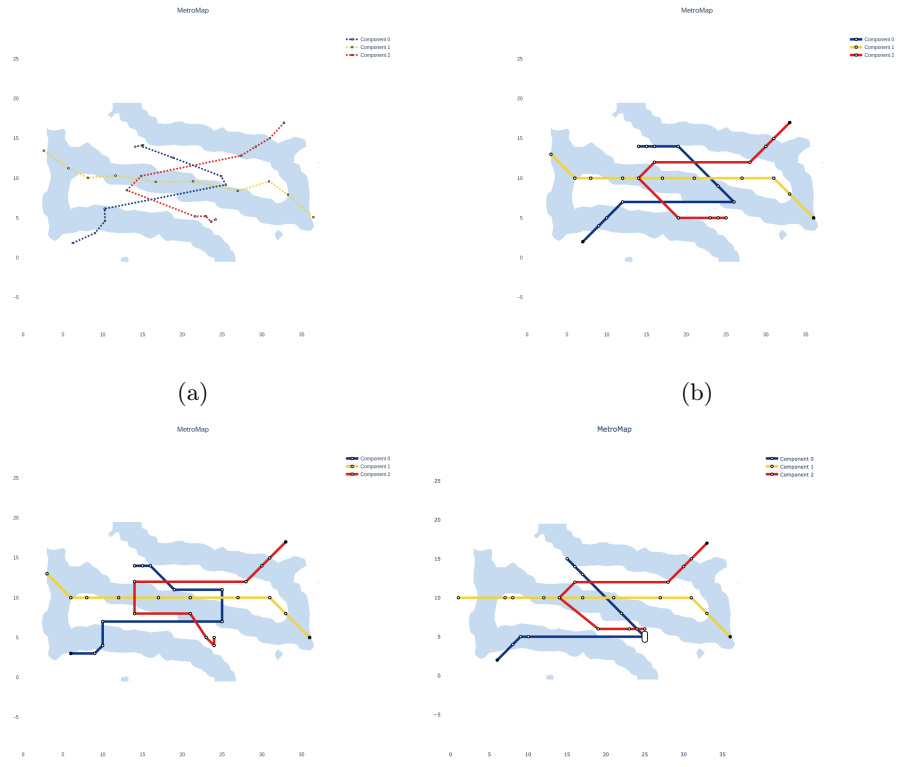


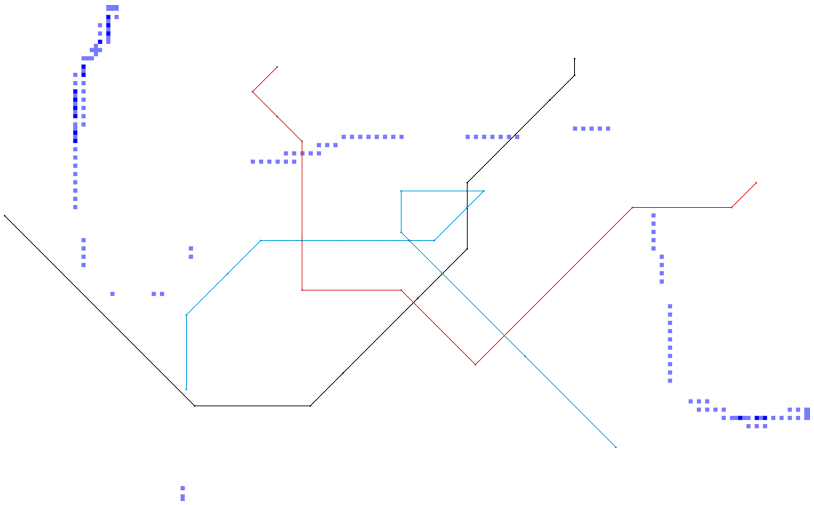
Figure 1: The SOM that is shown on the figures above was trained on the chainlink dataset using the JavaSOM toolbox. It had a size of 40x20 and for both visualizations we chose 10 bins. (a) shows the raw points that are obtained by the binning process, (b) shows the result of the MetroSolver with corner penalties applied and (c) shows the result with no corner penalties and (d) shows the result with extremely large corner penalties.

Looking at the figures above, we can clearly see that the overall layout of the original metro system (see Figure 1a) that was obtained from the binning step is preserved by the algorithm. When bends were penalized (penalty values: [0.0, 0.7, 1.4, 4.2, 5.6]), the solver favors a solution with mostly straight lines, and 135° angles between the stations (see Figure 1b). Only in one single instance the solver had to fall back to using a sharp angle of 45°. When the values of the penalty are set to zero, much more sharp corners become visible (see Figure 1c. This is especially evident at the lower end of the third component line (red) where the solution now contains a short segment at a sharp angle. The last image (Figure 1d) shows the solution when an extremely large penalty value is applied for each angle except straight lines. The resulting metro graph

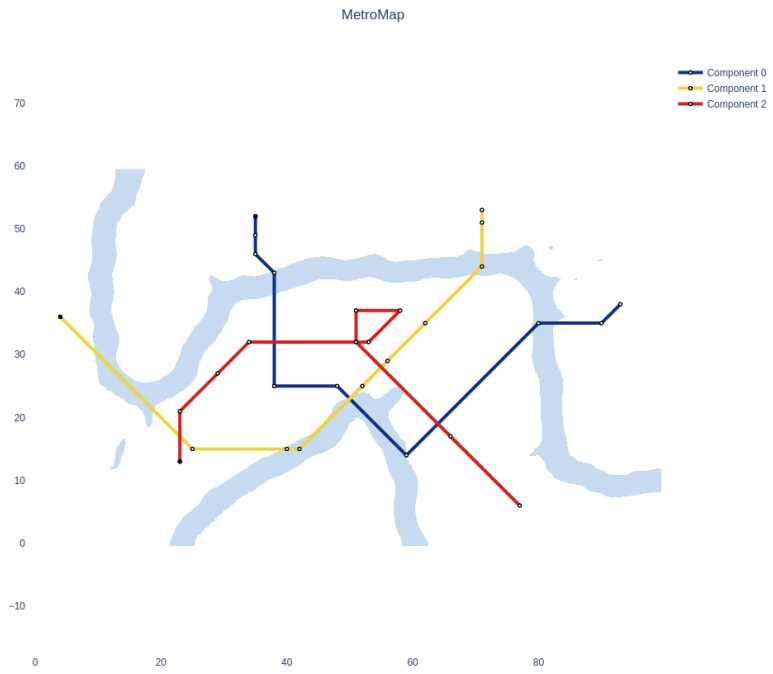
indeed shows even less bends than the one from Figure 1b, but the difference is not as pronounced as the difference between a penalty and no penalty. The high penalty values ([0, 20, 40, 40, 40]) straightened out one end of the second component line (yellow) and also one of the first component's line (blue). Further increasing the penalty values did not yield any notable difference in the appearance of the final metro map.

3 Comparison with the SOMToolbox

Because the origin of the coordinate system in the SOMToolBox is in the top left corner while the origin of the plotly-plots we used is in the bottom left corner, we flipped / mirrored the visualizations from the SOMToolBox vertically in order to compare them to the plots generated by our visualization.

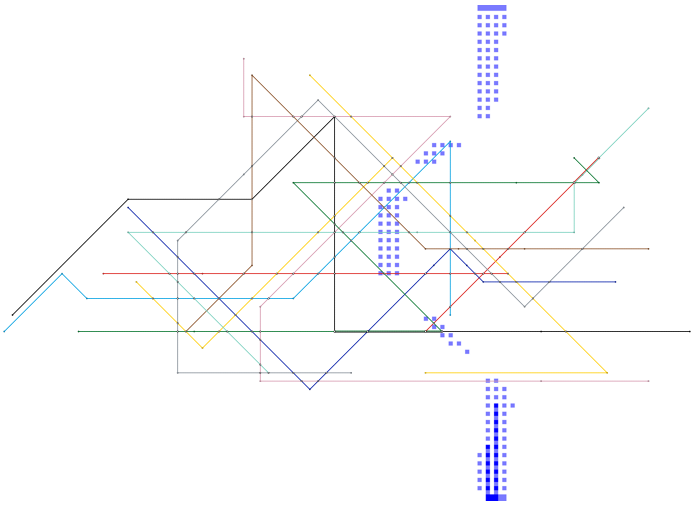


(a)

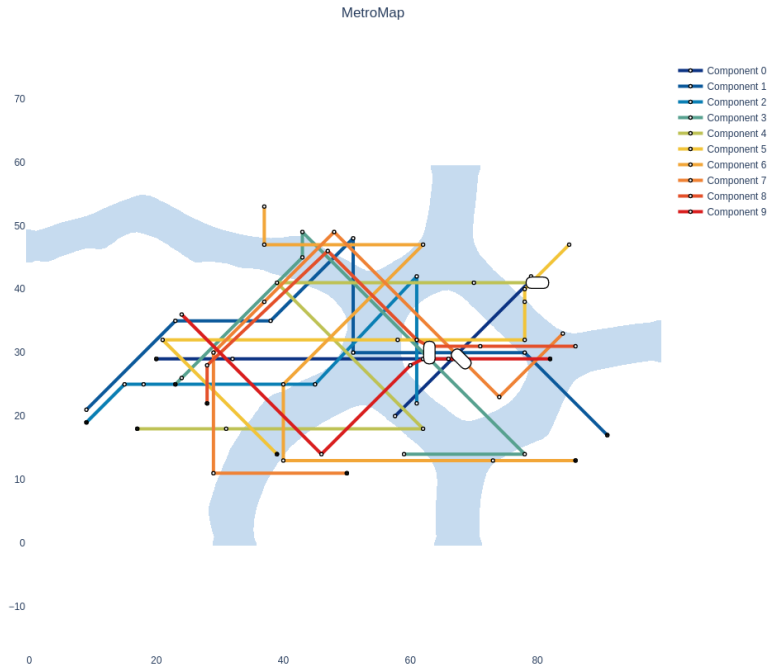


(b)

Figure 2: The two figures above show a SOM that was trained on the chainlink dataset using the JavaSOM toolbox. The SOM had a size of 100x60 and for both visualizations we chose 10 bins. The MetroMap visualizations were generated by (a) the Java SOMToolbox and (b) our algorithm.

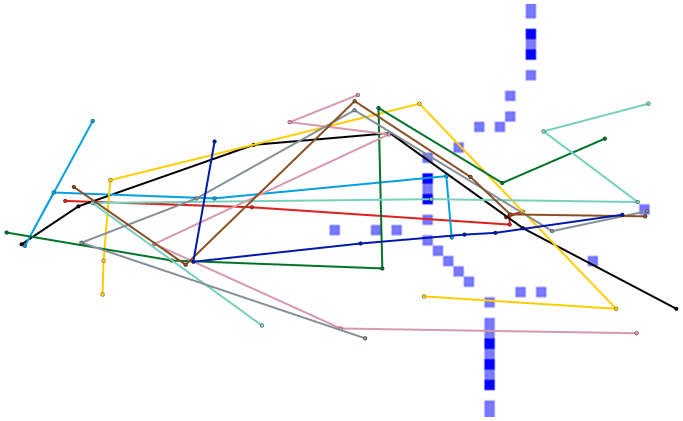


(a)

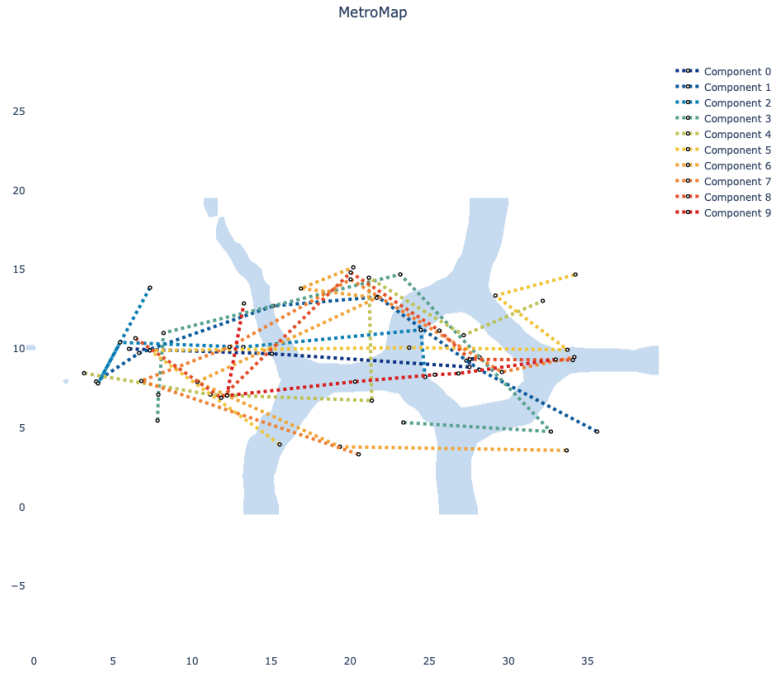


(b)

Figure 3: The two figures above show a SOM that was trained on the same 10clusters dataset using the JavaSOM toolbox. The SOM had a size of 100x60 and for both visualizations we chose 7 bins. The MetroMap visualizations were generated by (a) the Java SOMToolbox and (b) our algorithm. To enable good compairability with the SOMToolBox, we disabled corner penalties in this run.



(a)



(b)

Figure 4: The two figures above show a SOM that was trained on the 10clusters dataset using the JavaSOM toolbox. The SOM had a size of 40x20 and for both visualizations we chose 6 bins. The MetroMap visualizations were generated by (a) the Java SOMToolbox and (b) our algorithm. We disabled the snapping overlay in both visualizations to show the raw region centers.

In the figures above, we can clearly see that the results of our algorithm closely match the results of the Java SOMToolBox. While the raw / unsnapped region centers exactly match as seen in Fig. 4, we can observe some very subtle differences in the runs without corner penalties (Fig. 3) and some more slightly more noticable ones when using our implementation with corner penalties (Fig. 2). Due to the adjustable water levels in our implementation, we can make structures more visible in the U-Matrix-overlay which are visualized as rivers and lakes. The applied smoothing makes them look even more like real water-bodies.

References

[1] Robert Neumayer et al. "The metro visualisation of component planes for self-organising maps". In: *2007 International Joint Conference on Neural Networks*. IEEE. 2007, pp. 2788–2793.