Enhanced Spatial Arrangement of TreeMap Using Self-Organizing Maps

John Cannon

April 22, 2009

1 The Problem

The TreeMap algorithm provides a 2-dimensional view of a hierarchy of data in which the relative size of each item or group of items can be seen. The items are by nature grouped together in the visualization based on their membership in the tree. The original algorithm was very simple in the way it laid out the items. The items were laid out in the order they are in the tree, alternating the direction of division by level. This serves the purpose of relating the relative sizes and the group membership of each item, but the arrangement of the items within the group does not signify anything except the node order in the tree.



Figure 1: A simple TreeMap layout

Modifications to this algorithm have attempted to order the nodes but simply place them in a one dimensional ordering based on the original order or ordering on one component value [5, 1]. These types of simple sorting are not sufficient for arranging groups of multi-dimensional data. Other modifications break from the uniform division by attempting to make each rectangle have the lowest possible aspect ratio [2]. While this makes the tree more visually appealing, the arrangement still means little.

In this paper, a technique is described that attempts to arrange the nodes into an orientation that is indicative of their relationship in multidimensional space. The technique employed is based on the self-organizing map, or SOM for short. The SOM is a type of neural-network that dimensionally quantizes multidimensional vector spaces. In other words, it takes a set of points in N-dimensional space and maps them into a space of smaller dimensionality while attempting to preserve the topology. This is the basis for the solution. The algorithm was adjusted a little to allow for directing points to certain areas in the map. The basis of this algorithm is taken from an algorithm called SmartJitter which was originally described by Marjan Trutschl in "Intelligently resolving point occlusion" for two-dimensional scatter plots, but it has also been extended into many other forms [6]. An even more highly modified SOM algorithm is used to map the nodes from a multidimensional space into a constrained 2-dimensional space. This attempts to arrange the nodes such that the nodes closest to each other in the output are more similar than those farther away.

Once the nodes are arranged in 2-dimensional space, the problem is reduced to that of approximating the same layout for their corresponding rectangles in the TreeMap. The nature of the TreeMap limits the ability of the SOM to arrange the nodes. The TreeMap consists of nested rectangles representing the groups and subgroups of the data. The SOM can never move a node outside the bounds of its parent node. Therefore it becomes a sort of tug-of-war between the best regional placement (provided by the TreeMap) and the best local placement (provided by the SOM).

2 Self-Organizing Maps

Teuvo Kohonen first proposed the self-organizing map in the early 1980s. As mentioned earlier, the algorithm quantizes multi-dimensional space, producing a map of the data into a lower number of dimensions [3]. It is a neural

network that is trained by unsupervised learning, meaning there is no goal state for the network. Typically you provide as many vectors as possible that represent the type of data you will map using the map, and the algorithm produces the best fitting map. This map could then be used to classify vectors not used in the training. However, in our case, we just want the best map for all of our data, so we feed it all of the vectors. An input vector is simply a single data record. For this algorithm, all data records must have the same dimensionality and all values must be numeric.

The algorithm is run with the full set of input vectors and the map is constructed algorithmically. The result will not always be the same unless all random number generation used in the algorithm uses the same seed. On top of that, the training is limited to a specific number of epochs or iterations over the data set. Different data sets may require longer training times. It is therefore impossible to validate with 100% accuracy that the output is perfectly correct. One can only determine whether the map produced correctly correlates items.

The algorithm is fairly simple if one thinks of the map not as a neural-network but as a grid of independent nodes. The nodes are arranged in a grid and therefore have a position that can be thought of in normal Cartesian coordinates. Each node has an associated weight vector that is the same dimensionality as the vectors in the input space. These nodes define the means for mapping vectors to the output space. After training, the vectors to be mapped can be compared to all the weight vectors in the output map and the position of the node whose weight vector is most similar is assigned to that input vector.

Training takes place in a defined number of iterations, known as epochs, each of which refines the output map. At the start of training, all the node weight vectors are assigned initial values. These can be purely random values or specially chosen values. In our case, the minimum and maximum values for each component are used as a range for random values for the corresponding vector components. For each epoch, two operations are performed for each input vector: winning node search and weight vector update. The input vectors are used for these two operations in a random order, but not using a vector more than once. In our terminology, this is record scheduling. The schedule of records is random for each epoch. The search and update operations for a vector must be completed before the next scheduled record can be handled. Once all the epochs are completed, the nodes then comprise the output map.

The winning node search operation involves finding the best matching node for an input vector. This is sometimes called the BMU. This step involves comparing the current input vector against all the weight vectors in the map and choosing the one that is the most similar to the input vector. The idea of similarity here may vary and any number of metrics may be used to determine the best match. In most cases the Euclidean distance is taken for the two vectors and the node that has the weight vector with the lowest distance from the input vector is selected as the winning node. In the case of an exact tie, the first one found is chosen. An exhaustive search is required since there is no way to tell if another node in the map will be a better match. Once the winning node has been located, the weight vector update can begin.

The weight vectors are updated within a neighborhood around the winning node so that the association of the input vector with that node is strengthened. The vectors of very near neighbors are adjusted to look more like the input vector. The amount of the adjustment is based on the distance between the node being updated and the winning node (the grid distance, i.e. the Euclidean distance between the two node positions). Note that this is NOT a distance between weight vectors. The component values of the weight vector of the node currently being updated are modified by adding a scaled difference between the component value and the value of the corresponding component in the current input vector. This scaled difference is referred to in this text as the effect of the input vector on that vector component.

The product of the evaluation of a learning rate function and a neighborhood function yield the scaling factor for the adjustment. The learning rate function depends on the current epoch number and two parameters to the algorithm that define the minimum and maximum learning rate factors. The factor chosen decreases from epoch to epoch, which reduces the effect of vectors on the map as training progresses. The neighborhood function is a Gaussian function that depends on the size of the map and the current epoch number and reduces the effect of a vector exponentially as you move away from the winning node. At the first epoch, all of the vectors are affected, but as training continues, the radius of the area affected decreases according to the neighborhood function. All of the equations mentioned here can be seen below.

$$d_j^2 = \|V - w\| \Rightarrow c = min\{d_j\}$$

Equation 1. Finding the Winning Node.

$$w_j(t+1) = w_j(t) + \beta(k) \cdot g(k) \cdot q_j(t)$$
$$q_j(t) = w_j(t) - C_j$$

Equation 2. Updating the nodes.

β = Learning rate at epoch k
g = Neighborhood function at epoch k
d = Distance between the input vector V and the weight vector w_j
j = Current output node
C = Current winning node

$$\beta(k) = \beta_{initial} \cdot \left(\frac{\beta_{final}}{\beta_{initial}}\right)^{\frac{k}{k_{max}}}$$

Equation 3. Learning rate.

$$\sigma(k) = \|N\| \cdot \left(\frac{0.2}{\|N\|}\right)^{\frac{k}{k_{max}}}$$

Equation 4. Neighborhood Radius.

$$g(k) = e^{\left(\frac{-d_j(t)^2}{\sigma(k)^2}\right)}$$

Equation 5. Neighborhood weight function.

After all of the nodes in the map are updated for the current input vector, the next scheduled vector is chosen and the process is repeated. Once the input vector schedule is exhausted (i.e., all the vectors have been handled), the next epoch begins with a new record schedule. Note that the neighborhood and learning rate functions will now produce different values. Once all epochs have been completed, the mapping of the input vectors may be performed

by simply doing the winning node search for each vector and recording the position of the winning node without updating the map. At that point, the recorded position is the position of the data point in the output space.

The number of epochs used for training is a parameter to the algorithm and is specified by the user. The number required for a good map often depends on the data set. Some require many epochs while some are trained easily in a small number of epochs. The more epochs, the more stable the map usually is. We have in general defaulted to about 1000 epochs but have seen nice results with small data sets using as few as 50 epochs. It is a tradeoff between processing time and quality of the result.

3 TreeMap

The original TreeMap algorithm was first introduced in 1992 by Ben Shneiderman [4]. The idea is to start with a single rectangle representing the entire tree and then divide that rectangle into proportionate sub-rectangles for each child node. This process continues recursively down the tree, alternating the direction of subdivision at each level. This is a very simple and, for many purposes, very effective way of visualizing a tree. It demonstrates the grouping of the data and the size of the subgroups. This algorithm itself makes no attempt to order or arrange the nodes based on their content. It merely gives a visual representation of the tree.

4 TreeMapSOM

The new algorithm, TreeMapSOM, consists of two parts: SOM positioning and rectangle assignment. The SOM algorithm has been highly modified. It is similar in some respects to the SmartJitter algorithm. Both map each vector to defined search regions (a directed SOM). In SmartJitter, dividing a 2-dimensional plot space into smaller sub grids creates the regions, and the data records are mapped scatter-plot style to the sub-grids. For the TreeMapSOM, the regions are the rectangles of a TreeMap visualization and the points are mapped to those regions based on their node membership.

The TreeMap presents an additional problem. Since creating the node rectangles is the goal of the algorithm, the required regions are unknown at the start. The SOM is used to position the centers of each node for

assignment. Since we know the root rectangle, the SOM can be run in steps, organizing at each level to provide points for the rectangle division in order to create the regions for the next level. This new type of SOM is not only "directed" but also "iterative". The training takes place in multiple sessions that are followed by changing the set of regions–a process that is dependent on organization of the records between sessions. The following is a high level outline of the modified SOM process:

1. Start with one region (the whole visualization space)
2. Train one iteration
3. Organize the data records
4. Find the average organized coordinate of the records in each node on the next level
5. Divide this level's rectangles using the coordinates from 4.
6. Assign the rectangles to the nodes in the next level and set the regions on the SOM
7. If this isn't the last level, go back to 2.

The rectangle division takes a rectangle and a set of points that should lie in the rectangle and attempts to divide the rectangle into a set of smaller rectangles that are arranged in an approximation of the original orientation (see Figure 2). This is achieved through a recursive, trial and error method. The division starts in the direction of the longest axis of the points dividing out as many non-overlapping rectangles as possible. Any points that result in overlapping rectangles in this direction are added to a list, which is then recursively attempted in the direction of the longest axis of the subset. If neither method can create non-overlapping rectangles, the rectangles are simply assigned in sorted order along their longest dimension.

The following is an outline of the rectangle division algorithm:

1. Start with a base rectangle for the whole map and the center of the root node
2. If there is only one center, assign the base rectangle and return

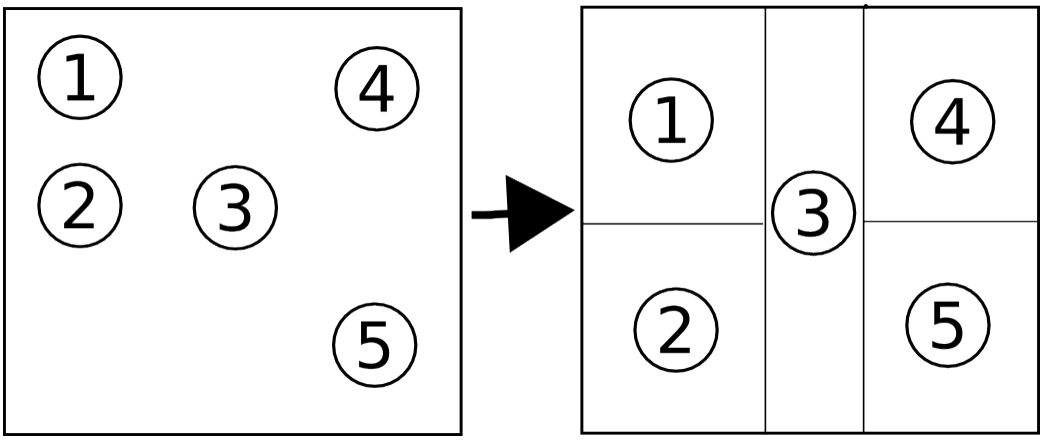


Figure 2: Example of assignment of 5 equal sized nodes from their SOM-provided center positions

3. Choose to divide on the axis with the largest range
4. Scale the center coordinates to have minimum of 0, maximum of 1.
5. Sort centers on longest axis
6. Add the first rectangle to the subdivide list
7. For each remaining center
   (a) Form the combined rectangle for the entire subdivide list and this center
   (b) If this center is contained in the combined rectangle
      i. Add to the subdivide list
   (c) Else
      i. Break out of loop
8. If subdivide list is not empty
9. Call this function recursively with the subdivide list
10. Else
    (a) Layout the rectangles in order (impossible to arrange)

11. Subtract the combined rectangle

12. If there are remaining centers in the input list, call this recursively to assign

5 Results

Few multi-dimensional hierarchical data sets could be located for testing, and what few could be found were in numerous different non-standard and faulty formats. Therefore a synthetic dataset generation algorithm was implemented for testing. The algorithm simply creates a tree with random numbers of branches at each level and creates data vectors with random component values that fall within ranges that are supposed to simulate clustering. For the puropose of reproducibility, the algorithm uses a hardcoded random number generator seed. A simple parser for the TreeMap '97 file format was also written to facilitate manually created datasets and also to allow testing with the 3 example datasets included with the application. The parser was created using the JJTree/JavaCC parser generator tools.

The results can best be seen using the interactive application developed along with the algorithm. The application provides a function to color the nodes based on their similarity (Euclidean distance) to a selected node as well as a line plot of vectors in the style of a parallel coordinates plot. With a sufficient grid size to arrange the data points, fair arrangements can be achieved.

When looking at the similarity of neighboring clusters, limitations can be seen where the tree structure has forced nodes into positions that do not correspond with the goals of the SOM-enhanced version of the TreeMap (Figure 4). The strict grouping of the original form just does not allow some nodes to be colocated. This can also be the fault of bad clustering or incorrect data dimension selection. If for example, the data is clustered on three of twelve dimensions, the other 9 dimensions could be pretty much randomly distributed among the leaf clusters in the tree. This would limit the SOM's ability to differentiate groups in the data and cause the algorithm to break down.

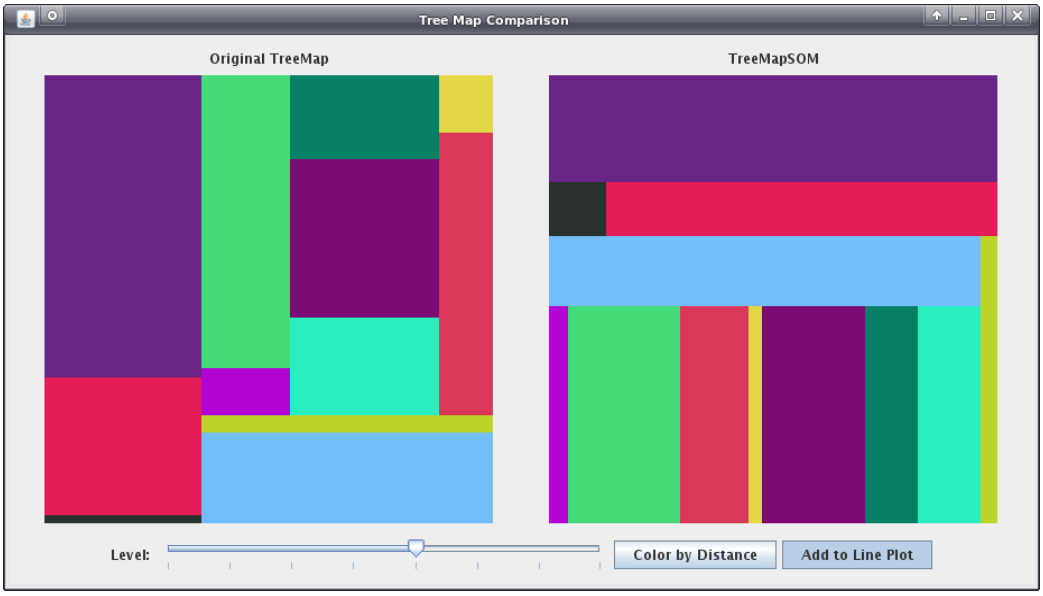


Figure 3: Side-by-side comparison (TreeMap on left, TreeMapSOM on right)

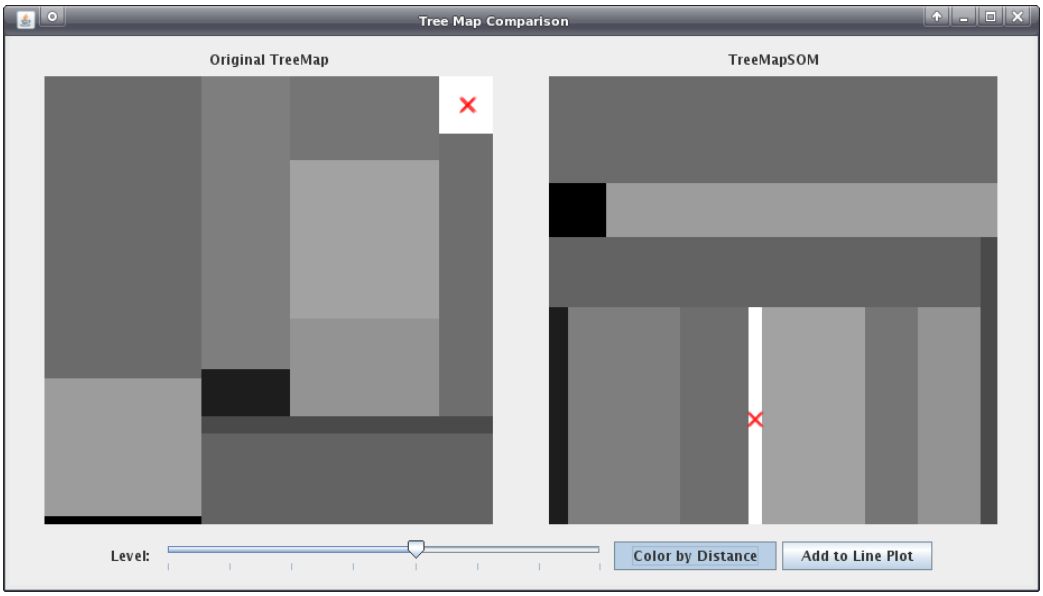


Figure 4: Nodes colored by dissimilarity to selected node (X marks selected)

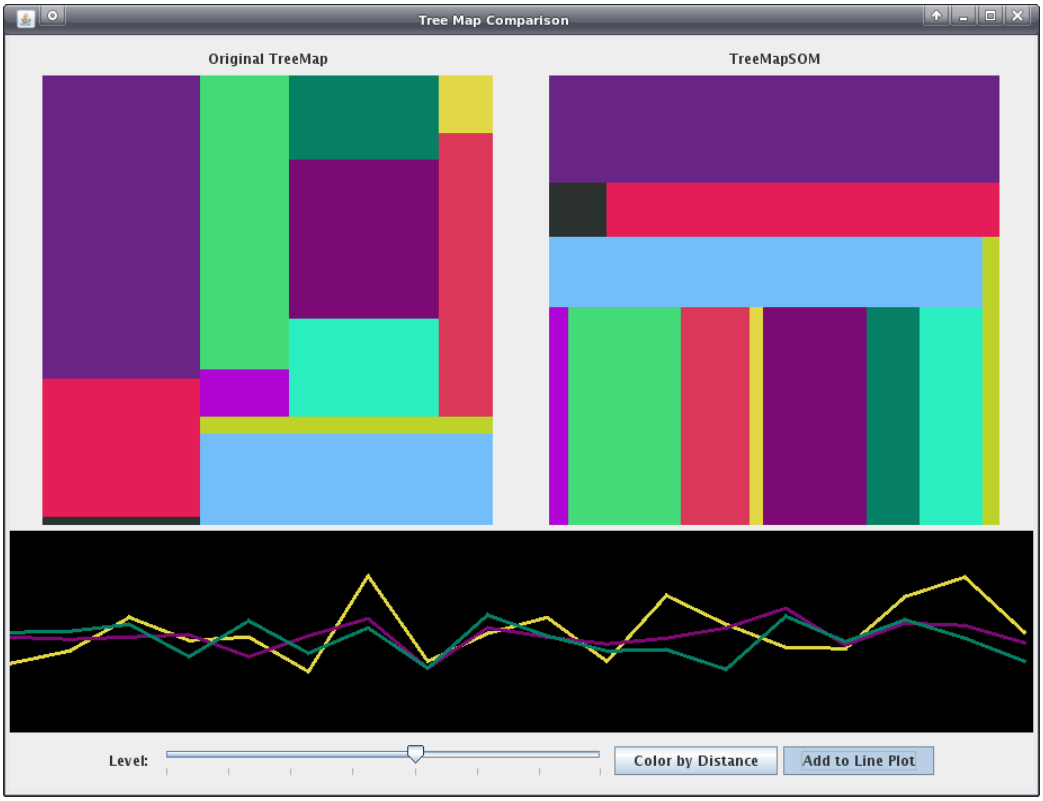


Figure 5: Line plot comparison of three nodes' centroid vectors (color coded)

References

[1] B. B. Bederson, B. Shneiderman, and M. Wattenberg. Ordered and quantum treemaps: Making effective use of 2d space to display hierarchies. *ACM Trans. Graph.*, 21(4):833–854, 2002.

[2] M. Bruls, K. Huizing, and J. J. V. Wijk. Squarified treemaps. In *In Proceedings of the Joint Eurographics and IEEE TCVG Symposium on Visualization*, pages 33–42. Press, 2000.

[3] T. Kohonen. Self-organizing mappings for two-dimensional (visual) display of high-dimensional pattern spaces. In P. Johansen and P. W. Becker, editors, *Proc. 3SCIA, Scand. Conf. on Image Analysis*, pages 35–41, Lund, Sweden, 1983. Studentlitteratur.

[4] B. Shneiderman. Tree visualization with tree-maps: 2-d space-filling approach. *ACM Trans. Graph.*, 11(1):92–99, 1992.

[5] B. Shneiderman and M. Wattenberg. Ordered treemap layouts. In *INFOVIS '01: Proceedings of the IEEE Symposium on Information Visualization 2001 (INFOVIS'01)*, page 73, Washington, DC, USA, 2001. IEEE Computer Society.

[6] M. Trutschl, G. Grinstein, and U. Cvek. Intelligently resolving point occlusion. *2003 IEEE Symposium on Information Visualization (InfoVis 2003)*, page 17, 2003.