188.413 Self-Organizing Systems Assignment 3 2021 WS Topic h1 Three Different Visualizations

Group h1:

- Philipp Hochhauser 01527619
- Fabian Nagler 11776817
- Sebastian Steiner 11777731

Link to repository: https://github.com/xHotch/SOS2021_ThreeVisualizations (https://github.com/xHotch/SOS2021_ThreeVisualizations)

In [1]:

```
import numpy as np
import holoviews as hv
from holoviews import opts
from holoviews import dim
from somtoolbox import SOMToolbox
from minisom import MiniSom
from SOMToolBox_Parse import SOMToolBox_Parse
import networkx as nx
from math import sqrt
hv.extension('bokeh')
#Global parameters

"""
Thresholds for Cluster connections
"""
cluster_t1 = 0.5
cluster_t2 = 1.0
cluster_t3 = 2.0

m_small = 40
n_small = 20

m_large = 100
n_large = 60
```

Parsing input data

In [2]:

```
# Read data from Java SOMToolbox
dataset = {}

#Load Chainlink and 10 Cluster data from http://www.ifs.tuwien.ac.at/dm/somtoolbox/datasets
data_10clusters = SOMToolBox_Parse("datasets\\10clusters\\10clusters.vec").read_weight_file
data_chainlink = SOMToolBox_Parse("datasets\\chainlink\\chainlink.vec").read_weight_file()

#Read small Test weights for comparison from http://www.ifs.tuwien.ac.at/dm/somtoolbox/data
weights_10clusters = SOMToolBox_Parse("datasets\\10clusters\\10clusters.wgt").read_weight_f
classes_10clusters = SOMToolBox_Parse("datasets\\10clusters\\10clusters.cls").read_weight_f
```

Training SOMs

Here we train two different SOMs (40x20, 100x60) for each of the input Datasets

In [3]:

```
#TODO learning rate, sigma etc.
som_10cluster_small = MiniSom(m_small, n_small, 10, sigma=2.0, learning_rate=0.7)
som_10cluster_small.train(data_10clusters['arr'], 1000)

som_10cluster_large = MiniSom(m_large, n_large, 10, sigma=3.0, learning_rate=0.6)
som_10cluster_large.train(data_10clusters['arr'], 1000)

som_chainlink_small = MiniSom(m_small, n_small, 3, sigma=0.8, learning_rate=0.6)
som_chainlink_small.train(data_chainlink['arr'], 10000)

som_chainlink_large = MiniSom(m_large, n_large, 3, sigma=0.8, learning_rate=0.6)
som_chainlink_large.train(data_chainlink['arr'], 10000)
```

Activity Histograms

Activity Histogramms visualize the distance between one given datapoint of the input data and all weight vectors. For measuring the distance between one input vector and the weight vectors, we use Euclidean distance.

The function *calculateActivityHist* takes the input vector, which shall be projected onto the SOM and calculates the activity histogram with respect to the given weight vector.

In order to retrieve a visualization which is comparable to the visualizations retrieved from the Java implementation of the SOM toolbox, we added the functionality to calculate a custom color gradient. The corresponding color values were taken from the Java implementation [1].

[1] SOMToolbox, IFS Tu Wien, http://www.ifs.tuwien.ac.at/dm/somtoolbox/index.html (http://www.ifs.tuwien.ac.at/dm/somtoolbox/index.html)

In [4]:

```
def calculateColorGradientMap(colorGradient, colorCount):
  colorMap = []

  for i in range(1, len(colorGradient)):
    ((prevR,prevG,prevB), prevP) = colorGradient[i-1]
    ((r,g,b), p) = colorGradient[i]

    lowerBorder = prevP * colorCount
    upperBorder = p * colorCount

    for j in range(0, int(upperBorder-lowerBorder)):
      currentWeight = j / (upperBorder-lowerBorder)
      prevWeight = 1.0 - currentWeight

      newR = int(prevR * prevWeight + r * currentWeight)
      newG = int(prevG * prevWeight + g * currentWeight)
      newB = int(prevB * prevWeight + b * currentWeight)

      colorMap.append((newR, newG, newB))

  colorMap.append((r,g,b))

  return colorMap

activatyHistColorGradient = [((0,0,128), 0.0), ((0,0,255), 0.15), ((0,128,255), 0.35), ((24
activatyHistColorCount = 255

activativityHistColorMap = calculateColorGradientMap(activatyHistColorGradient, activatyHis

def calculateActivityHist(_m, _n, _weights, _ivector):
    hist = np.zeros(_m * _n)

    for i in range(0, _m * _n):
        weight_vector = _weights[i]
        hist[i] = np.sqrt(np.sum(np.power(weight_vector-_ivector,2)))

    return hist.reshape(_m, _n)
```

Minimum Spanning Tree

For our minimum spanning tree visualization we use the paper [1] for the high level overview and the Java reference implementation [2] for specific details.

For the minimum spanning tree on weights, we iterate through our model size and build a fully connected tree. The weights of each edge are given by the euclidean distance of the corresponding weight vectors. After constructing this graph, we apply a minimum spanning tree algorithm to obtain our final graph. Optionally, we re-calculate the edge weights for display in the output visualization.

The two options available are:

- show_som_weights: this setting controls from which data to construct the MST. If set to True it will use all units, if set to False it will use only those matched to the input data.

- weight_lines: this setting controls whether or not to use weight as basis for line thickness in the visualization.

Sources:

[1] Mayer, Rudolf, and Andreas Rauber. "Visualising Clusters in Self-Organising Maps with Minimum Spanning Trees." International Conference on Artificial Neural Networks. Springer, Berlin, Heidelberg, 2010.

[2] SOMToolbox, IFS Tu Wien, http://www.ifs.tuwien.ac.at/dm/somtoolbox/index.html (http://www.ifs.tuwien.ac.at/dm/somtoolbox/index.html)

In [5]:

```
def rescale(x,b, a, minx, maxx):
    scale = (b-a)*((x-minx)/(maxx-minx))+a
    return scale
def distance(w1, w2):
    """
    Returns the euclidean distance for two arbitrarily large vectors

    """
    return sqrt(sum([(x-y)**2 for x, y in zip(w1, w2)]))

def MST(_m, _n, _weights, _idata, _show_som_weights, _weight_lines):
    """
    Calculates all line segments to be displayed for the chosen MST.

    _m and _n         represent the size
    _weights          is the unit data
    _idata            represents the input data
    _show_som_weights base the MST on the SOM weights instead of the unit data
    _weight_lines     whether to set thickness based on weight
    """

    # reshape the weights array to obtain it in size m x n
    w = _weights.reshape(_m, _n, len(_weights[0]))

    # build the MST for SOM weights
    if _show_som_weights:
        G_weights = nx.Graph()

        # iterate through all 2D coordinates twice
        for row1 in range(0, _m):
            for col1 in range(0, _n):
                for row2 in range(0, _m):
                    for col2 in range(0, _n):
                        # fully connect the graph based on the pair-wise distance function
                        G_weights.add_edge((row1, col1), (row2, col2), weight = distance(w[
        # shrink graph to get edge list in MST
        T_weights = nx.minimum_spanning_edges(G_weights)

    # build the MST for unit vectors
    if not _show_som_weights:
        G_data = nx.Graph()

        matching = []

        # In this step we need to map our input data to the corresponding units.
        # For this we approximate by using a greedy nearest neighbour setting.
        for _d in _idata:
            best_distance = float("inf")
            best = None
            for row in range(0, _m):
                for col in range(0, _n):
                    d = distance(_d, w[row, col])
                    if best_distance > d:
                        best_distance = d
                        best = (row, col, w[row, col])
            matching.append(best)

        # iterate through whole matching twice
        for (row1, col1, w1) in matching:
```

```
            for (row2, col2, w2) in matching:
                # fully connect the graph based on pair-wise distance function
                G_data.add_edge((row1, col1), (row2, col2), weight = distance(w1, w2))
        # shrink graph to get edge list in MST
        T_data = nx.minimum_spanning_edges(G_data)

    # create arrays in which to store output data for visualization
    startx = []
    starty = []
    endx = []
    endy = []
    w_out = []
    max_weight = 0

    # append all line segments
    if _show_som_weights:
        for ((col1, row1), (col2, row2), data) in T_weights:
            startx.append(col1)
            endx.append(col2)
            starty.append(row1)
            endy.append(row2)
            # get maximum weight for proper scaling
            if _weight_lines:
                w_out.append(1/data['weight'])
                if data['weight'] > max_weight:
                    max_weight = data['weight']
            else:
                w_out.append(5)

    # append all line segments
    if not _show_som_weights:
        for ((col1, row1), (col2, row2), data) in T_data:
            startx.append(col1)
            endx.append(col2)
            starty.append(row1)
            endy.append(row2)
            # get maximum weight for proper scaling
            if _weight_lines:
                w_out.append(1/data['weight'])
                if data['weight'] > max_weight:
                    max_weight = data['weight']
            else:
                w_out.append(5)

    if _weight_lines:
        w_out = [x * max_weight * 0.15 for x in w_out]

    startx = [rescale(x,-0.5,0.5,0,_m) for x in startx]
    starty = [rescale(y,-0.5,0.5,0,_n, flip = True) for y in starty]
    endx = [rescale(x,-0.5,0.5,0,_m) for x in endx]
    endy = [rescale(y,-0.5,0.5,0,_n, flip = True) for y in endy]

    return starty, startx , endy, endx, w_out

def visualizeMST(cc):
    return hv.Segments(cc,['x', 'y', 'x1', 'y1'], 'w').opts(line_width=dim('w'), colorbar=T
```

Cluster Connections

To implement the Cluster connections, we referred to the description provided in [1], as well as the Java implementation provided in [2].

Algorithm Overview: Cluster Connections are a post-processing step, applied after SOM Training.

- First we calculate a distance matrix containing pairwise distances of the weight vectors of the trained SOM, using np.linalg.norm function.
- We then iterate over all the cells of the SOM.
  - We draw a connection to the cell on the right, if the distance is smaller than some threshold
  - We draw a connection to the cell below, if the distance is smaller than some threshold

(Three thresholds are used to determine the color of the drawn connections. If the distance is greater than all the thresholds, no connections will be drawn. Indexing the unit_distance matrix is a bit tricky, as it has a different shape than the output SOM. The get_unit_index function returns the index for the unit distance matrix for a cell x, y in the SOM. (Similar to the Java implementation in [2])

Usage:

Use calculate calculate functions to create starting and endpoints for segments. Use visualize function to create hv.Segments from the list of points.

Sources:

[1] Merkl, Dieter, and Andreas Rauber. "Alternative ways for cluster visualization in self-organizing maps." Workshop on Self-Organizing Maps. 1997.

[2] SOMToolbox, IFS Tu Wien, http://www.ifs.tuwien.ac.at/dm/somtoolbox/index.html (http://www.ifs.tuwien.ac.at/dm/somtoolbox/index.html)

In [6]:

```
def rescale(x,b, a, minx, maxx, flip = False):
    scale = (b-a)*((x-minx)/(maxx-minx))+a
    if flip:
        scale = scale * -1
    return scale

def get_color(distance):
    """
    Returns a color, depending on thresholds.

    Returns None if the input value is higher than all the thresholds.
    """
    if distance < cluster_t1:
        return '#000000'
    elif distance < cluster_t2:
        return '#696969'
        # return "#5555FF"
    elif distance < cluster_t3:
        return '#C0C0C0'
    else:
        return None

def get_unit_index(x, y, xSize):
    """
    Returns the Index of a Unit for a given x and y.

    Is used to index the symmetric unit distance matrix.
    """
    return y * xSize + x;

def calculate(_m, _n, _weights):

        #Calculate matrix containing pair-wise distances from the weights of the som, shep
        unit_distance = np.linalg.norm(_weights[:, None, :] - _weights[None, :, :], axis=-1


        # Array of x Position from the start of the Segments
        startx = []
        # Array of y Position from the start of the Segments
        starty = []
        # Array of x Position from the end of the Segments
        endx = []
        # Array of y Position from the end of the Segments
        endy = []
        # Array of Colors for each of the Segments
        c = []

        #Loop over SOM cells
        for col in range(0,_m):

            for row in range(0,_n):

                #add horizontal lines
                if (col<_m-1):

                    #Get distance from the current cell to the cell on the right
                    d = unit_distance[get_unit_index(col,row,_m),get_unit_index(col+1,row,_
                    color = get_color(d)
```

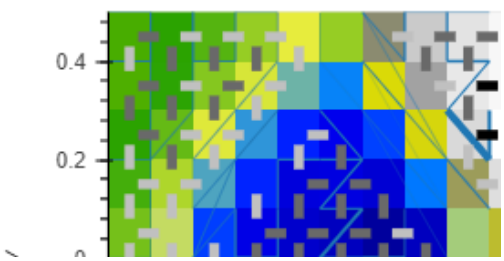
```
                    #We add some offset to the start and end position of the segments
                    if color:
                        startx.append(col+0.7)
                        endx.append(col+1.2)
                        starty.append(row+0.5)
                        endy.append(row+0.5)
                        c.append(color)

                #add vertical lines
                if (row<_n-1):

                    #Get distance from the current cell to the cell below
                    d = unit_distance[get_unit_index(col,row,_m),get_unit_index(col,row+1,_
                    color = get_color(d)

                    #We add some offset to the start and end position of the segments
                    if color:
                        startx.append(col+0.5)
                        endx.append(col+0.5)
                        starty.append(row+0.7)
                        endy.append(row+1.2)
                        c.append(color)

        #Return calculated arrays
        startx = [rescale(x,-0.5,0.5,0,_m, flip = True) for x in startx]
        starty = [rescale(y,-0.5,0.5,0,_n) for y in starty]
        endx = [rescale(x,-0.5,0.5,0,_m, flip = True) for x in endx]
        endy = [rescale(y,-0.5,0.5,0,_n) for y in endy]
        return startx, starty, endx, endy, c
        # return starty, startx , endy, endx, c

def visualizeCC(cc):
    ccImage = hv.Segments(cc,['x', 'y', 'x1', 'y1'], 'c').opts(color=dim('c'), colorbar=Tru
    return ccImage
```

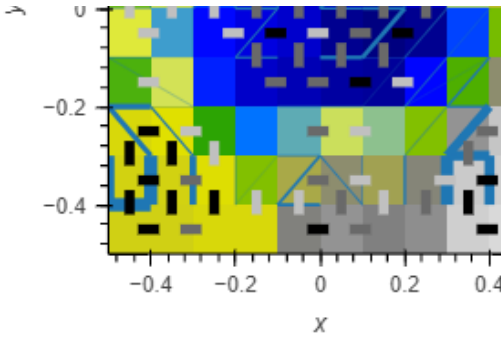
Visualization

In [7]:

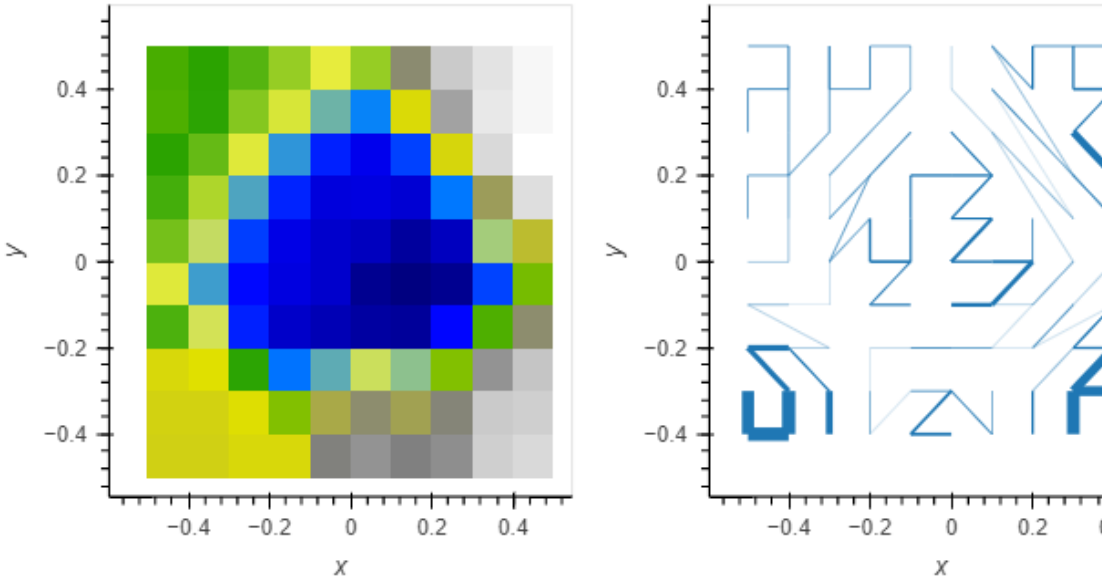
```
# Load data into the variables from a wgt file
wts = SOMToolBox_Parse("datasets\\10clusters\\10clusters.wgt").read_weight_file()
wtdata = weights_10clusters['arr']

# Alternative: Load data into the variables from a trained minisom object, and reshape acco
"""
wtdata = som_10cluster_small._weights.reshape(-1,10)
idata = data_10clusters['arr']
"""

# Load vector input data
idata = data_10clusters['arr']

#Optional: Change thresholds for cluster connections
cluster_t1 = 0.5
cluster_t2 = 1.0
cluster_t3 = 2.0

#Calculate Cluster Connections
cc = calculate(10, 10, wtdata)
ccImage = visualizeCC(cc)

#Calculate Activity Histogram
activityHist = calculateActivityHist(10, 10, wtdata, idata[0])
activityHistImage = hv.Image(activityHist).opts(cmap = activativityHistColorMap)

#Calculate Minimum Spanning Tree
mst = MST(10, 10, wtdata, idata, True, True)
mstImage = visualizeMST(mst)

print("Displaying the layers stacked")
display(activityHistImage * mstImage * ccImage)
print("Displaying the layers side-by-side")
display(activityHistImage + mstImage + ccImage)
#TODO combine visualization

cluster_t1 = 4
cluster_t2 = 6
cluster_t3 = 10
cc = calculate(10, 10, wtdata)
ccImage = visualizeCC(cc)
print("Too high Threshold values will cause the image to be overfilled with connections")
display(ccImage)

print("MST calculated from mapping Unit Data to Neurons")
mst = MST(10, 10, wtdata, idata, False, False)
mstImage = visualizeMST(mst)
display(mstImage)
```

```
Displaying the layers stacked
```

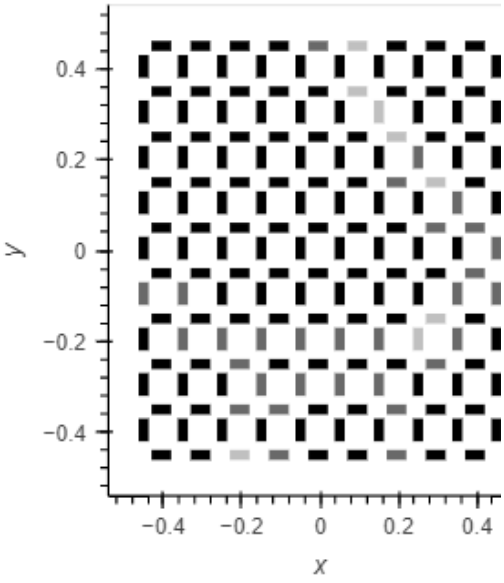




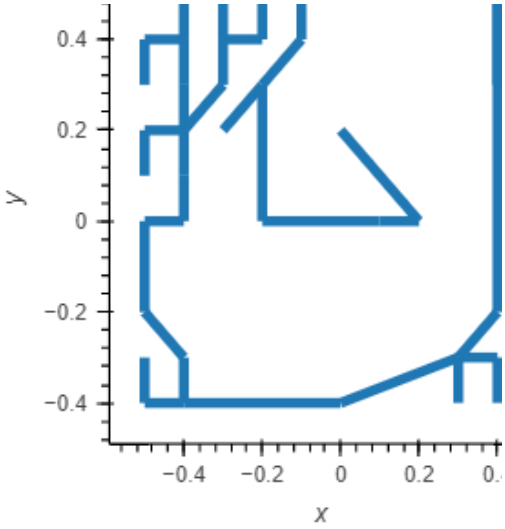
Displaying the layers side-by-side



Too high Threshold values will cause the image to be overfilled with connect ions



MST calculated from mapping Unit Data to Neurons

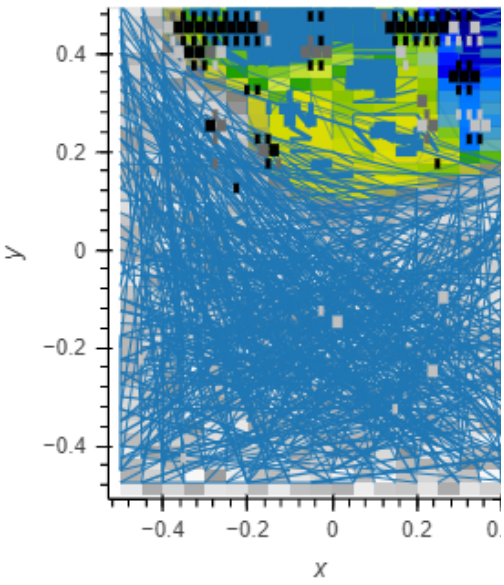


In [ ]:

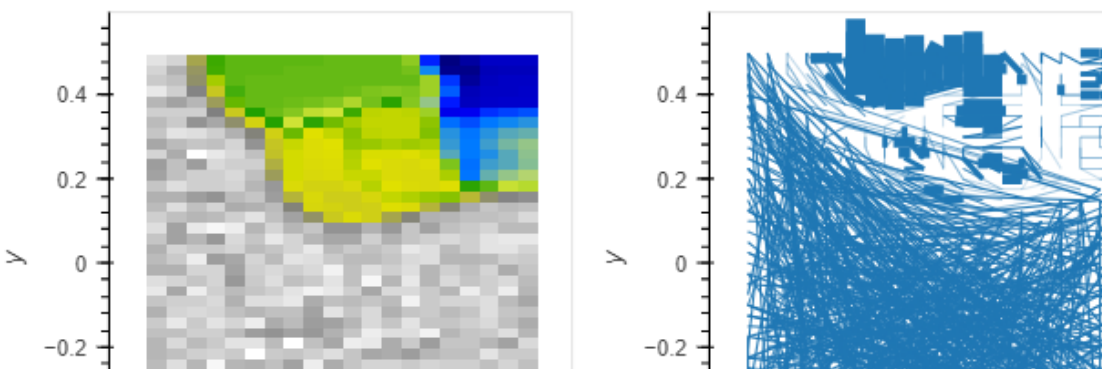
In [8]:

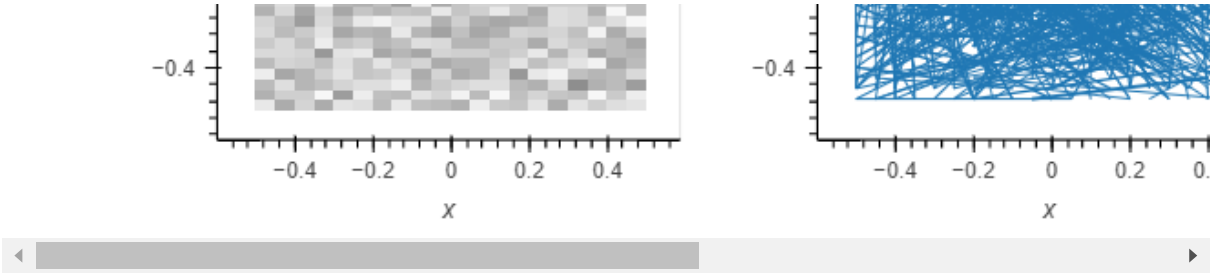
```
# Change parameters of Cluster Connections
cluster_t1 = 0.2
cluster_t2 = 0.4
cluster_t3 = 0.7

# Load calculated SOM
wtdata = som_10cluster_small._weights.reshape(-1,10)
idata = data_10clusters['arr']

cc = calculate(m_small, n_small, wtdata)
ccImage = visualizeCC(cc)

activityHist = calculateActivityHist(m_small, n_small, wtdata, idata[0])
activityHistImage = hv.Image(activityHist).opts(cmap = activativityHistColorMap)

mst = MST(m_small, n_small, wtdata, idata, True, True)
mstImage = visualizeMST(mst)

print("Displaying the layers stacked")
display(activityHistImage * mstImage * ccImage)
print("Displaying the layers side-by-side")
display(activityHistImage + mstImage + ccImage)
```

```
Displaying the layers stacked
```

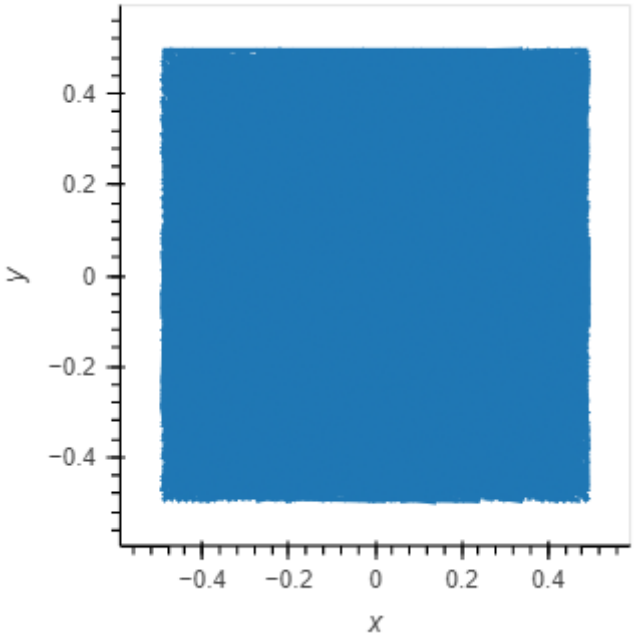



```
Displaying the layers side-by-side
```





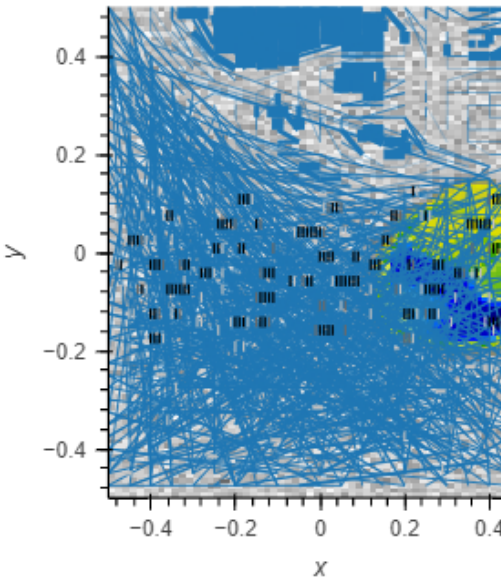
We disabled the MST for the large maps, as there was no real information gain and our implementation was became really slow with it:



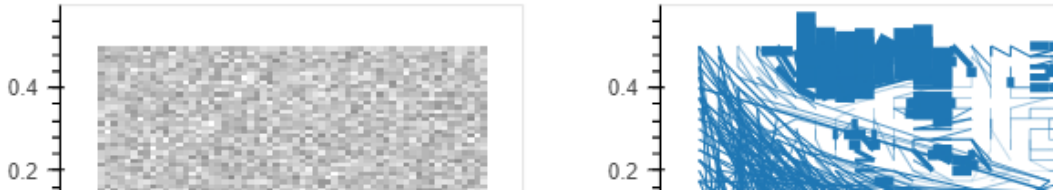
In [9]:

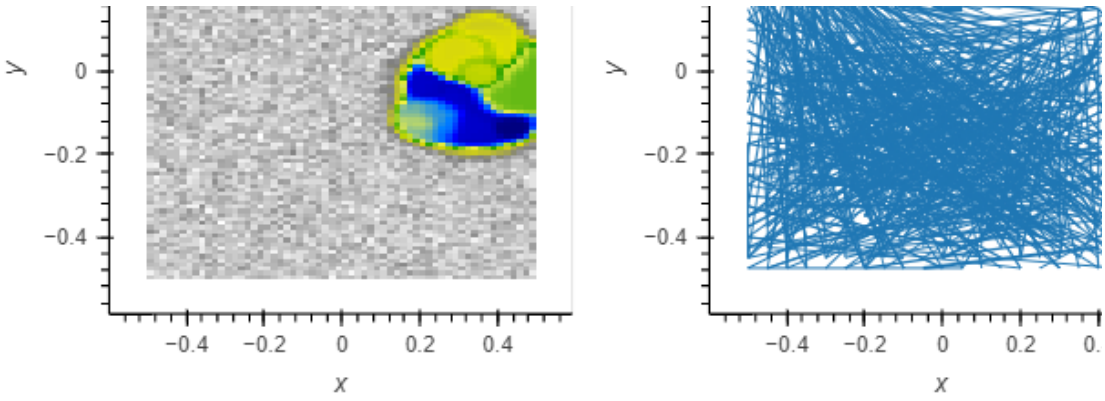
```
# Change parameters of Cluster Connections
cluster_t1 = 0.2
cluster_t2 = 0.4
cluster_t3 = 0.7

# Load calculated SOM
wtdata = som_10cluster_large._weights.reshape(-1,10)
idata = data_10clusters['arr']

cc = calculate(m_large, n_large, wtdata)
ccImage = visualizeCC(cc)

activityHist = calculateActivityHist(m_large, n_large, wtdata, idata[0])
activityHistImage = hv.Image(activityHist).opts(cmap = activativityHistColorMap)

# We disabled the MST for the large maps
"""
mst = MST(m_large, n_large, wtdata, idata, True, True)
mstImage = visualizeMST(mst)
"""


print("Displaying the layers stacked")
display(activityHistImage * mstImage * ccImage)
print("Displaying the layers side-by-side")
display(activityHistImage + mstImage + ccImage)
#TODO combine visualization
```

```
Displaying the layers stacked
```




```
Displaying the layers side-by-side
```

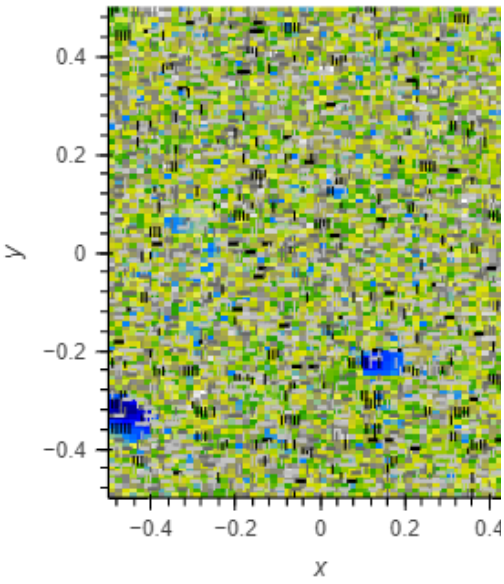



y
0
−0.2
−0.4
−0.4
−0.2
0
0.2
0.4
x
y
0
−0.2
−0.4
−0.4
−0.2
0
0.2
0.
x

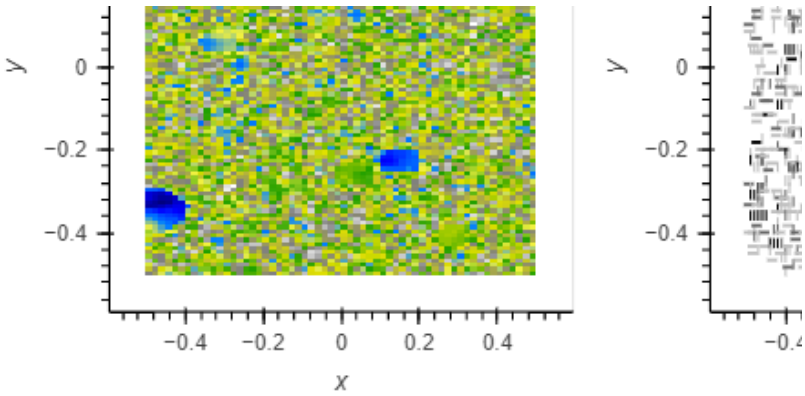
In [10]:

```
# Change parameters of Cluster Connections
cluster_t1 = 0.2
cluster_t2 = 0.4
cluster_t3 = 0.7

# Load calculated SOM
wtdata = som_chainlink_large._weights.reshape(-1,3)
idata = data_chainlink['arr']

cc = calculate(m_large, n_large, wtdata)
ccImage = visualizeCC(cc)

activityHist = calculateActivityHist(m_large, n_large, wtdata, idata[0])
activityHistImage = hv.Image(activityHist).opts(cmap = activativityHistColorMap)

# We disabled the MST for the large maps
"""
mst = MST(m_large, n_large, wtdata, idata, False, False)
mstImage = visualizeMST(mst)
"""


print("Displaying the layers stacked")
display(activityHistImage * ccImage)
print("Displaying the layers side-by-side")
display(activityHistImage + ccImage)
#TODO combine visualization
```

```
Displaying the layers stacked
```



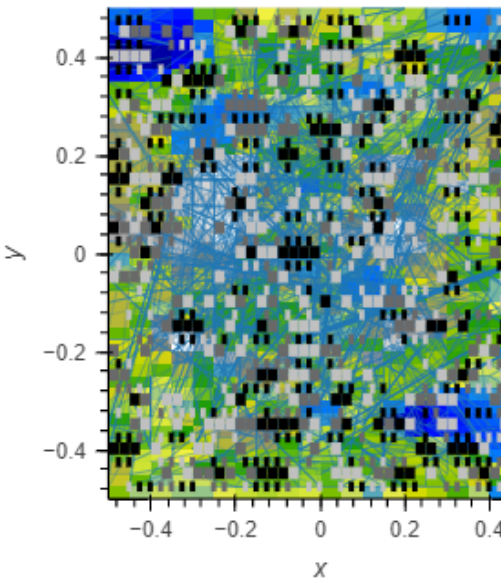

```
Displaying the layers side-by-side
```


y
0
−0.2
−0.4
−0.4
−0.2
0
0.2
0.4
x
y
0
−0.2
−0.4

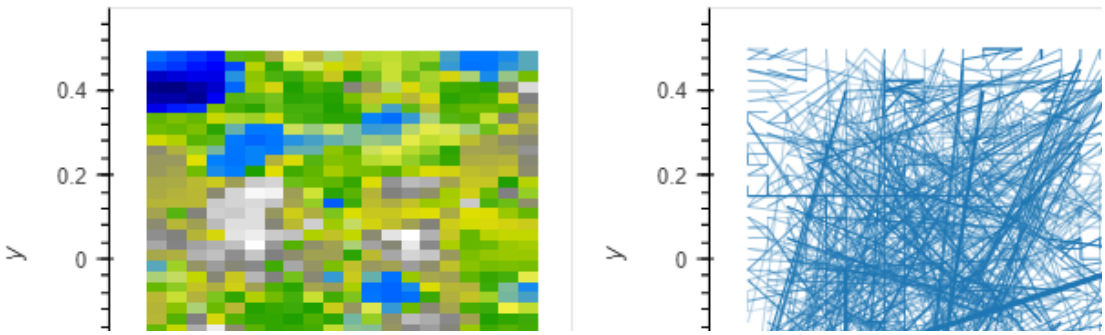
In [11]:

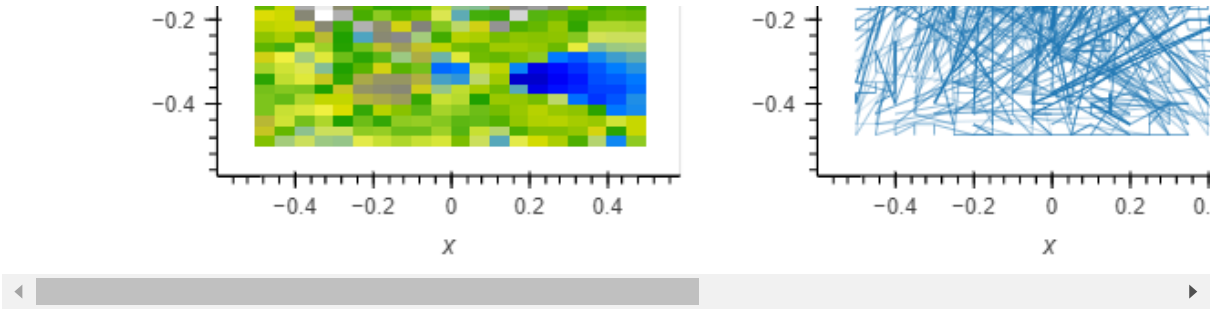
```
# Change parameters of Cluster Connections
cluster_t1 = 0.2
cluster_t2 = 0.4
cluster_t3 = 0.7

# Load calculated SOM
wtdata = som_chainlink_small._weights.reshape(-1,3)
idata = data_chainlink['arr']

cc = calculate(m_small, n_small, wtdata)
ccImage = visualizeCC(cc)

activityHist = calculateActivityHist(m_small, n_small, wtdata, idata[0])
activityHistImage = hv.Image(activityHist).opts(cmap = activativityHistColorMap)

mst = MST(m_small, n_small, wtdata, idata, True, True)
mstImage = visualizeMST(mst)

print("Displaying the layers stacked")
display(activityHistImage * mstImage * ccImage)
print("Displaying the layers side-by-side")
display(activityHistImage + mstImage + ccImage)
#TODO combine visualization
```

```
Displaying the layers stacked
```




```
Displaying the layers side-by-side
```



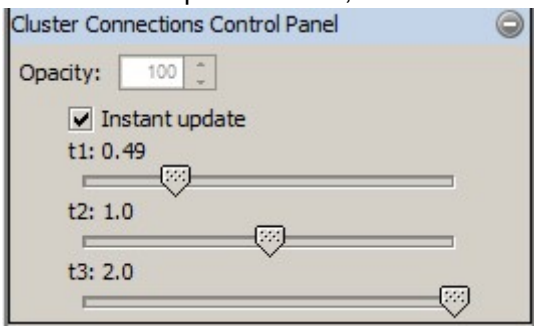


Comparison with SOM Toolbox

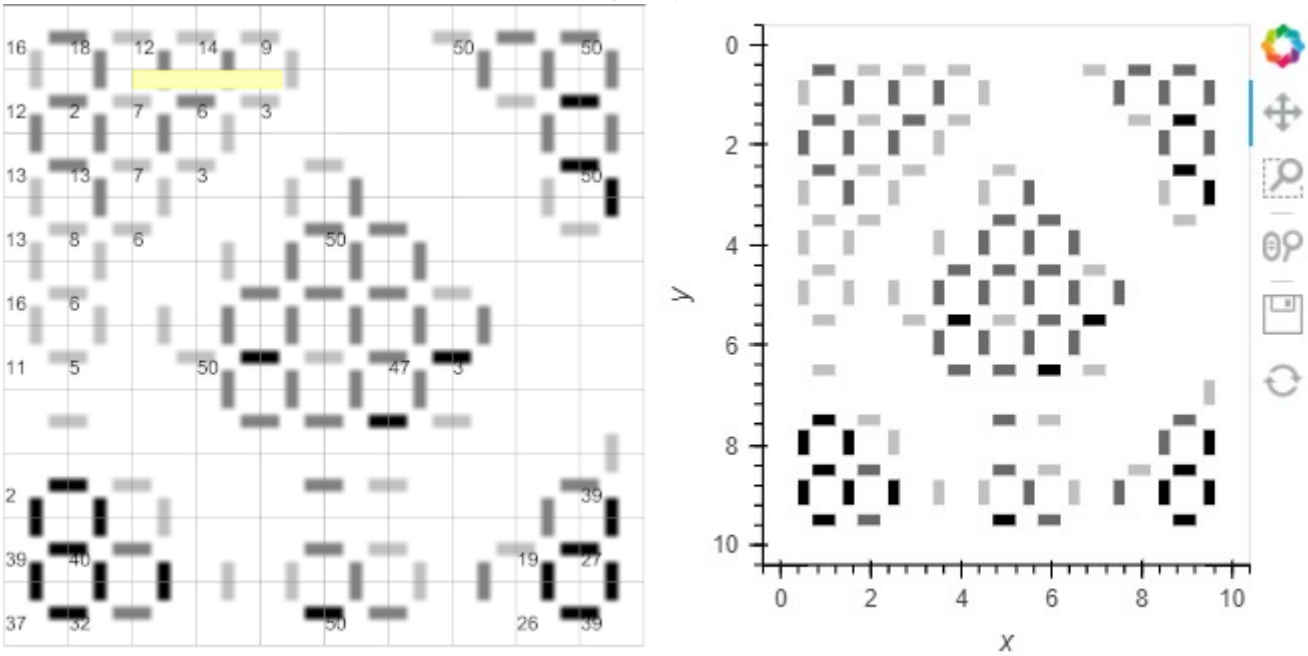
We compared the Visualizations with a small (10x10) pretrained SOM taken from the 10 clusters dataset (http://www.ifs.tuwien.ac.at/dm/somtoolbox/datasets.html (http://www.ifs.tuwien.ac.at/dm/somtoolbox/datasets.html)).

Cluster Connections

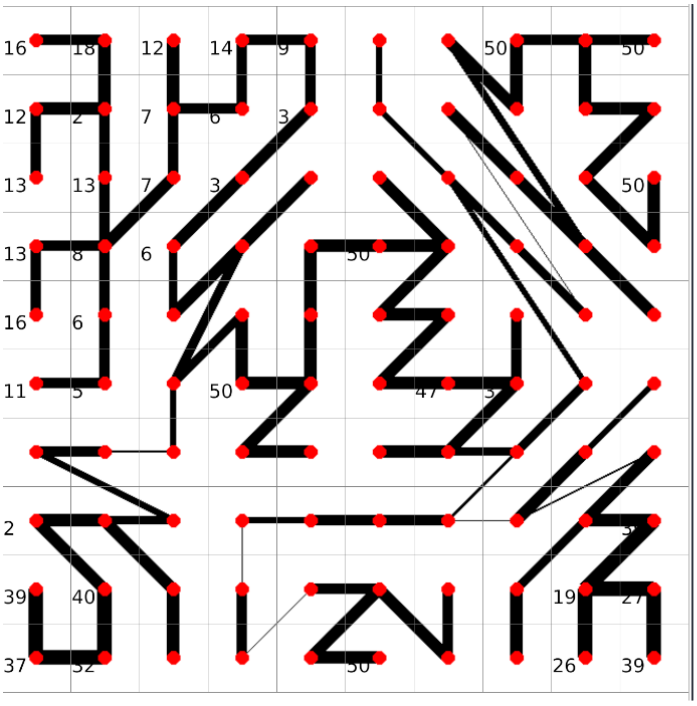
As the implementation was based on the Java implementation, the results look almost identical. Parameters:

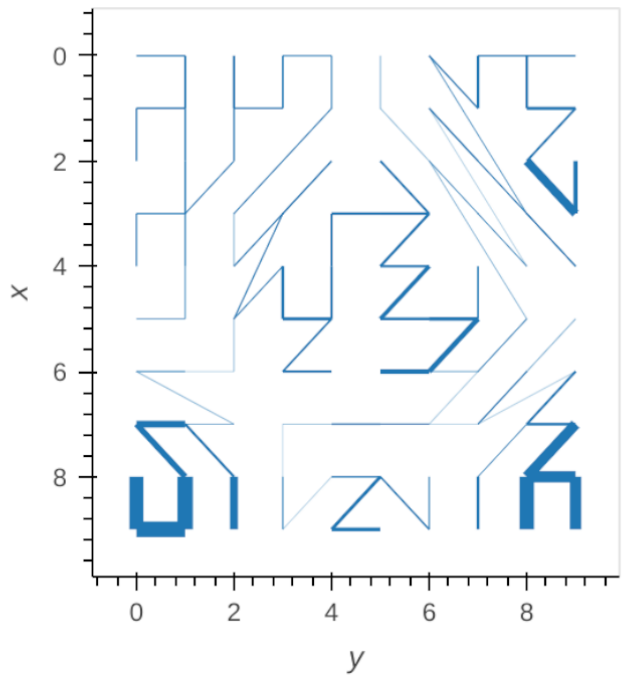


Comparison of Cluster Connections Left Java - Right Python

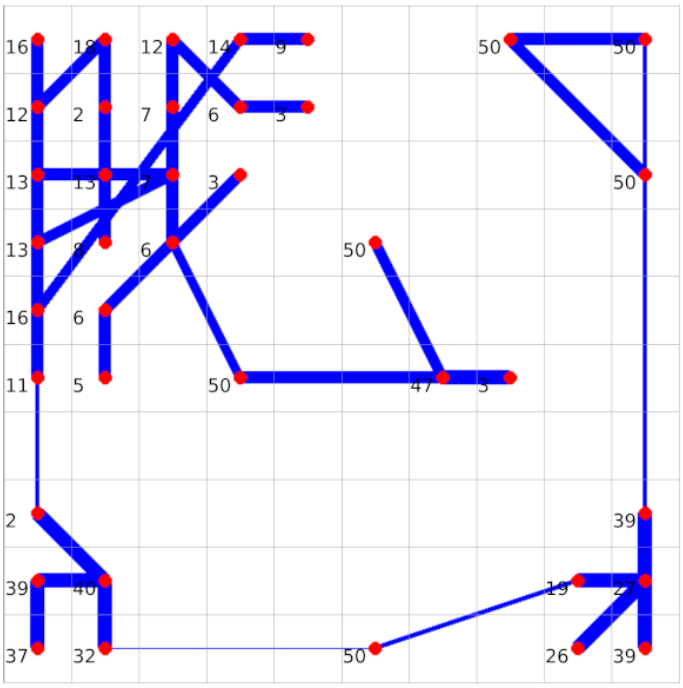


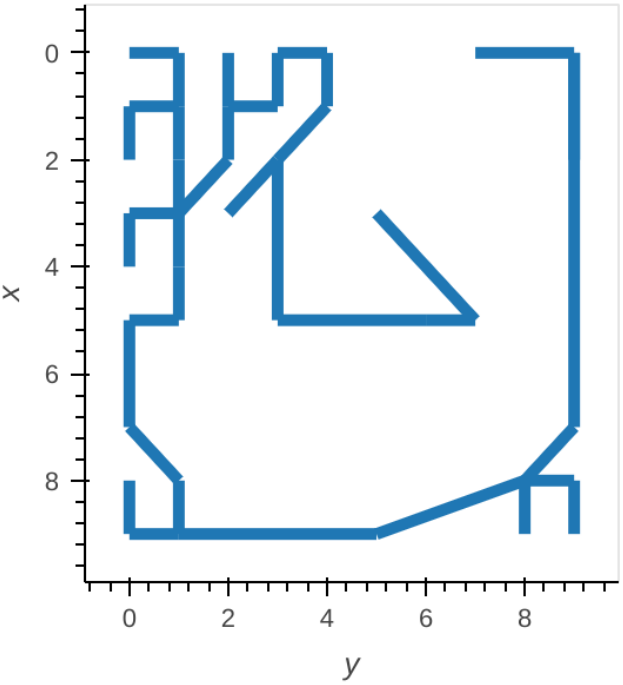
Comparison of MST: Left Java - Right Python



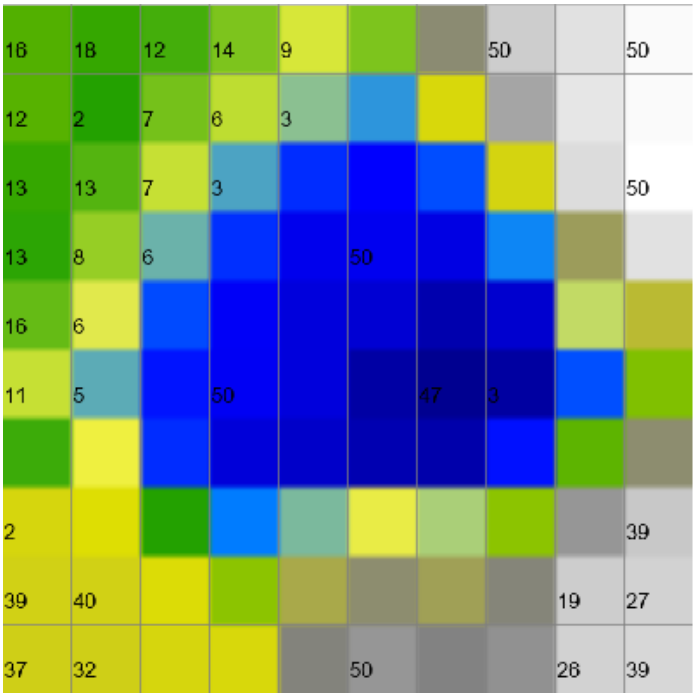


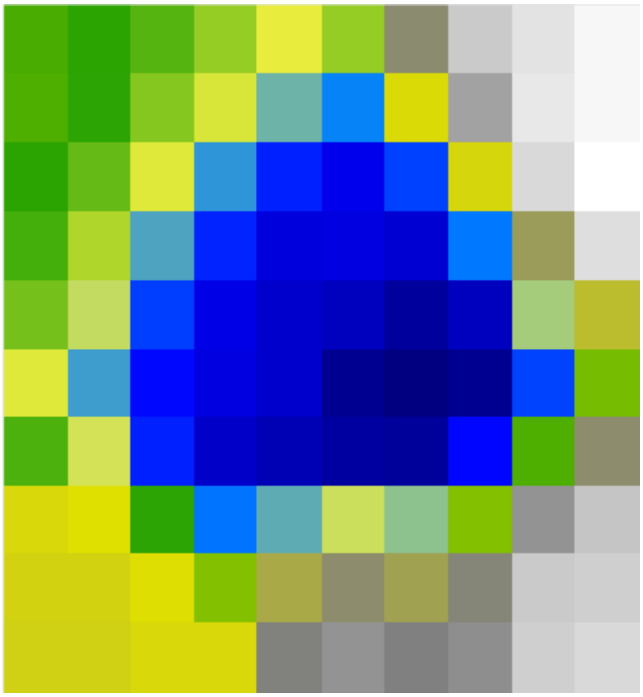
Comparison of MST: Left Java - Right Python





Comparison of Activity Histogram Left Java - Right Python


16 18 12 14 9 50 50
12 2 7 6 3
13 13 7 3 50
13 8 6 50
16 6
11 5 50 47 3
2 39
39 40 19 27
37 32 50 26 39



In [ ]: