Self-organizing Map – Untangler

This is a partial listing of the Java code for the Self-organizing map program working on untangling data from a square set of data points.

It includes all the code from the **Initialize** and **Evolution** sections, but does not include the variables' descriptions or the display section. The specification of the variables, for example the type of each variable and reasonable dimensions for the arrays, should be clear from the context.

Initialization section

```
mapsize=20;
nhood=4.0; // Starting number of neighbors to the target cell

/* The number of neighbors is given as a real number, since, although the actual
number of neighbors is an integer, nhood decreases by ndrop each cycle as the
neighborhood shrinks. */

ndrop=0.0005;

/* Speed determines how fast the simulation runs. The display is updated at the end of
each pass through the "Evolution" section of code; this code is executed "speed"
times. If speed is a larger number, many loops are run before the display is updated
and the map converges rapidly. If speed is set to a small value (1 is the minimum) the
display is updated once every cycle and the map changes slowly. */

speed=1;

/* The amount by which the weights are adjusted each cycle is "adjustment". This is
reduced each cycle by the value of adjustsize so that the weights are perturbed by
progressively smaller amounts as the map evolves. */

adjustment=0.8;
adjustsize=0.0001;

/* If minnhoodisone is true, the smallest neighborhood includes the BMU and its four
closest neighbors. If this variable is set to false, the neighborhood includes the BMU
alone. */

minnhoodisone=true;

cycle=0;
```

```
/* The logical variables gaussian, mexhat and linear can be used to choose the type of
neighborhood function. In this version of the program only the linear function is
implemented. */

gaussian=false;
mexhat=false;
linear=true;

/* Initialize the network weights. */

for (i=0; i<mapsize; i++)
{
   for (j=0; j<mapsize; j++)
   {
      weights[i][j][0]=1+(mapsize-1)*Math.random();
      weights[i][j][1]=1+(mapsize-1)*Math.random();
   }
}
```

Evolution section

```
cycle++;

/* Reduce the neighborhood and size of "adjustment". */

nhood=nhood-ndrop;
if (nhood<0) nhood=0;
if (minnhoodisone && nhood<1.0) nhood=1.0;

adjustment=adjustment-adjustsize;
if (adjustment<adjustsize) adjustment=adjustsize;

/* Choose a random point {xa,ya}, where both xa and ya are in the range 0 to
(mapsize-1). This point comprises the input data for the map. */

for (int iloop=0; iloop<speed; iloop++)
{
   xa=(mapsize-1)*Math.random();
   ya=(mapsize-1)*Math.random();

/* Locate the Best Matching Unit {besti,bestj} */

   besti=0;
   bestj=0;
   bestdiff=500.0;  // Difference between weights and input data
   for (i=0; i<mapsize; i++)
   {
      for (j=0; j<mapsize; j++)
```

```
        {
          diff=Math.abs(xa-weights[i][j][0])+Math.abs(ya-weights[i][j][1]);
          if (diff<bestdiff)
          {
            bestdiff=diff;
            besti=i;
            bestj=j;
          }
        }
      }
```

/* Adjust the weights at the BMU */

```
    weights[besti][bestj][0]=weights[besti][bestj][0]*(1.0-adjustment)+adjustment*xa;
    weights[besti][bestj][1]=weights[besti][bestj][1]*(1.0-adjustment)+adjustment*ya;
```

/* Adjust the weights at neighboring units. The neighborhood around the BMU extends to nhood units on either side, so first set the limits over which the update will run. */

```
    int nhoodint;
    nhoodint=(int)nhood;
    if (nhoodint>0)
    {
```

/* Make sure the neighborhood doesn't extend beyond the map edges. */

```
        left=besti-nhoodint;
        if (left<0) left=0;
        right=besti+nhoodint;
        if (right>(mapsize-1)) right=mapsize-1;

        top=(bestj-nhoodint);
        if (top<0) top=0;
        bottom=bestj+nhoodint;
        if (bottom>(mapsize-1)) bottom=mapsize-1;

        for (i=left; i<=right; i++)
        {
          for (j=top; j<=bottom; j++)
          {
```

// Determine the distance between the BMU and the neighbour

```
            distance=Math.sqrt((double)((besti-i)*(besti-i)+(bestj-j)*(bestj-j)));
```

/* If a linear fall off is required, the weighting is distance/nhood. */

```
            double changefac;
            changefac=distance*adjustment/(double)(nhood);
```

```
            if (linear)
            {
               weights[i][j][0]=(weights[i][j][0]+changefac*xa)/(1.0+changefac);
               weights[i][j][1]=(weights[i][j][1]+changefac*ya)/(1.0+changefac);
            }
         }
      }
   }
} // End of loop for the new point

/* Copy weights into the x and y arrays for display. */

np=0;
for (i=0; i<mapsize; i++)
{
   for (j=0; j<mapsize; j++)
   {
      x[np]=weights[i][j][0];
      y[np]=weights[i][j][1];
      np++;
   }
}

/* Finally determine where the lines will be drawn on the map. */

nlines=0;

for (i=0; i<mapsize; i++)
{
   for (j=0; j<mapsize; j++)
   {
      istart=i-1;
      if (istart<0) istart=0;
      iend=i+1;
      if (iend>mapsize) iend=mapsize;
      jstart=j-1;
      if (jstart<0) jstart=0;
      jend=j+1;
      if(jend>mapsize) jend=mapsize;
      for (int iloop=istart; iloop<iend; iloop++)
      {
         for (int jloop=jstart; jloop<jend; jloop++)
         {
            if (iloop == i || jloop == j)
            {
               linex[nlines]=weights[i][j][0];
               liney[nlines]=weights[i][j][1];
               linedx[nlines]=-weights[i][j][0]+weights[iloop][jloop][0];
               linedy[nlines]=-weights[i][j][1]+weights[iloop][jloop][1];
               nlines++;
```

```
            }
        }
    }
}
```