Growing Cell Structures

This is a partial listing of the Java code for the Growing Cell Structures program.

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

```
nnodes=3;  // Number of nodes at the start

nlines=3;   // Number of lines that will connect the nodes

/* At each node is stored a vector of weights. In this simple illustration, the first two
weights will be x and y values and the position of the node in the display will be set
equal to those values. */

/* Notes on Display:

   The variables fed into the "Positions and Size" section of the Arrows imager
properties are the starting points of the arrows, in x, y and optionally z coordinates,
and the displacement of the other end of the arrow, so if a line was to be drawn from
(3,4) to (9,2), the X and Y values would be 3 and 4 respectively, while Size X and
Size Y would be 6 and -2. */

/* Set up the initial weights. */

for (int i=0; i<nnodes; i++)
{
   weights[i][0]=5.0+2.0*Math.random();
   weights[i][1]=5.0+2.0*Math.random();
}

weights[0][0]=5.0;
weights[0][1]=3.0;
weights[1][0]=4.0;
weights[1][1]=3.5;
weights[2][0]=1.0;
weights[2][1]=8.5;

/* The connectedto array tells us which nodes are neighbors and so will be connected
with lines on the display. */

connectedto[0][0]=1;
```

```
connectedto[0][1]=2;
connectedto[1][0]=0;
connectedto[1][1]=2;
connectedto[2][0]=0;
connectedto[2][1]=1;

/* nneighbours keeps track of how many neighbours each node has. */

nneighbours[0]=2;
nneighbours[1]=2;
nneighbours[2]=2;

errorsum[0]=0.0;
errorsum[1]=0.0;
errorsum[2]=0.0;

/* Zero the signal counter */

for (int i=0; i<nnodes; i++)
{
   signalcounter[i]=0;
}

cycle=0;
longcycle=0;
nodeaddfreq=400;
ed=0.005;
```

Evolution section

```
cycle++;
longcycle++;
errordecay=1.0-ed;

newnodex=weights[nnodes-1][0];
newnodey=weights[nnodes-1][1];

/* Generate a data point on the donut. */

theta=360.0*Math.random();
rx=5.0+0.5*Math.random()+4.0*Math.sin(theta);
ry=5.0+0.5*Math.random()+4.0*Math.cos(theta);

/* Pass through all units to see which set of weights most closely matches the sample
pattern. */

double mindiff=10000.0;
double diff=10000.0;
```

```
int bestnode=0;

for (int i=0; i<nnodes; i++)
{
   diff=(weights[i][0]-rx)*(weights[i][0]-rx)+(weights[i][1]-ry)*(weights[i][1]-ry);
   if (diff<mindiff)  // found a better match than any previous ones
   {
      mindiff=diff;
      bestnode=i;    // record the position of the best node so far
   }
}

/* Update the signalcounter for the best node */

signalcounter[bestnode]++;

/* Reduce the signalcounter for all nodes by a fixed proportion. */

for (int i=0; i<nnodes; i++)
{
   signalcounter[i]=signalcounter[i]*errordecay;
}

/* Adjust the weights at the best node node slightly. */

weights[bestnode][0]=weights[bestnode][0]*0.94+0.06*rx;
weights[bestnode][1]=weights[bestnode][1]*0.94+0.06*ry;

/* and adjust the weights at neighbouring nodes. */

for (int i=0; i<nneighbours[bestnode]; i++)
{
   neighbour1=connectedto[bestnode][i];
   weights[neighbour1][0]=weights[neighbour1][0]*0.998+0.002*rx;
   weights[neighbour1][1]=weights[neighbour1][1]*0.998+0.002*ry;
}

/* Finally update the errorsum for the node. */

errorsum[bestnode]=errorsum[bestnode]+mindiff;

/* Copy the weights vector into the x and y points vector so that the display will work.
*/

for (int i=0; i<nnodes; i++)
{
   nodex[i]=weights[i][0];
   nodey[i]=weights[i][1];
}
```

```
/* Determine the contents of the length arrays so that the connecting lines can be
drawn. This step is inefficient because each line is drawn twice (once from the first
point to the second and then back again). However, it is probably at least as quick to
do as to check each time a new line is identified whether it has already been noted. */

nlines=0;
for (int i=0; i<nnodes; i++)
{
   for (int j=0; j<nneighbours[i]; j++)
   {
      pointx[nlines]=nodex[i];
      pointy[nlines]=nodey[i];
      neighbour1=connectedto[i][j];
      lengthx[nlines]=-nodex[i]+nodex[neighbour1];
      lengthy[nlines]=-nodey[i]+nodey[neighbour1];
      nlines++;
   }
}

/* Periodically add an extra node. */

if (cycle>nodeaddfreq) // add a node section starts here...
{
   cycle=0;

/* A1. Identify the node with the greatest signal counter. */

   bestnode=0;
   bestsignal=signalcounter[0];
   for (int i=1; i<nnodes; i++)
   {
      if(signalcounter[i]>bestsignal)
      {
         bestsignal=signalcounter[i];
         bestnode=i;
      }
   }

/* A2. Find the neighbour of this node for which the difference in weights between it
and the best node is greatest. */

   bestneighbour=0;
   maxdiff=0.0;
   for (int i=0; i<nneighbours[bestnode]; i++)
   {
      neighbour1=connectedto[bestnode][i];
      double a=Math.pow((weights[bestnode][0]-weights[neighbour1][0]),2) +
Math.pow((weights[bestnode][1]-weights[neighbour1][1]),2);
      if (maxdiff<a)
      {
```

```
            bestneighbour=neighbour1;
            maxdiff=a;
        }
    }

/* A3. Insert a new node. First give the node the average weights of its neighbours. */

    weights[nnodes][0]=0.5*(weights[bestnode][0]+weights[bestneighbour][0]);
    weights[nnodes][1]=0.5*(weights[bestnode][1]+weights[bestneighbour][1]);

/* A4. Now connect it to the nodes between which it lies. */

    connectedto[nnodes][0]=bestnode;
    connectedto[nnodes][1]=bestneighbour;
    nneighbours[nnodes]=2;

/* A5. Remove connections between best node and bestneighbour. */

    for (int i=0; i<nneighbours[bestnode]; i++)
    {
        if (connectedto[bestnode][i]==bestneighbour)
        {
            if (i<nneighbours[bestnode]-1)
            {
                for (int j=i; j<nneighbours[bestnode]; j++)
                {
                    connectedto[bestnode][j]=connectedto[bestnode][j+1];
                }
            }
        }
    }
    nneighbours[bestnode]--;
    for (int i=0; i<nneighbours[bestneighbour]; i++)
    {
        if (connectedto[bestneighbour][i]==bestnode)
        {
            if (i<nneighbours[bestneighbour]-1)
            {
                for (int j=i; j<nneighbours[bestneighbour]; j++)
                {
                    connectedto[bestneighbour][j]=connectedto[bestneighbour][j+1];
                }
            }
        }
    }
    nneighbours[bestneighbour]--;

/* A6. Check which nodes both bestneighbour and bestnode are connected to. */

    ncommonnodes=0;
```

```
   for (int i=0; i<nneighbours[bestnode]; i++)
   {
      for (int j=0; j<nneighbours[bestneighbour]; j++)
      {
         if (connectedto[bestnode][i]==connectedto[bestneighbour][j])
         {
            commonnodes[ncommonnodes]=connectedto[bestnode][i];
            ncommonnodes++;
         }
      }
   }
```

/* A7. If the number of common nodes is not zero, add links from the new node to the common nodes. */

```
   if (ncommonnodes>0)
   {
      for (int i=0; i<ncommonnodes; i++)
      {
         connectedto[nnodes][nneighbours[nnodes]]=commonnodes[i];
```

/* And do the reciprocal link... */

```
         neighbour1=commonnodes[i];
         connectedto[neighbour1][nneighbours[neighbour1]]=nnodes;
         nneighbours[neighbour1]++;
         nneighbours[nnodes]++;
      }
   }
```

/* A8. Now add links between the bestnode, nestneighbour and the new node./ */

```
   connectedto[bestnode][nneighbours[bestnode]]=nnodes;
   connectedto[bestneighbour][nneighbours[bestneighbour]]=nnodes;
   nneighbours[bestnode]++;
   nneighbours[bestneighbour]++;
```

/* A9. Decrease the error variable for all neighbours of the new node by an amount related to the number of neighbours. */

```
   for (int i=0; i<nneighbours[nnodes]; i++)
   {
      neighbour1=connectedto[nnodes][i];
      errorsum[neighbour1]=errorsum[neighbour1]*(1.0-1.0/nneighbours[nnodes]);
   }
```

/* A10. Set the errorsum of the new node to be the average of the errors for the nodes to which is it connected. */

```
   errorsum[nnodes]=0.0;
```

```
    for (int i=0; i<nneighbours[nnodes]; i++)
    {
       errorsum[nnodes]=errorsum[nnodes]+errorsum[connectedto[nnodes][i]];
    }
    errorsum[nnodes]=errorsum[nnodes]/nneighbours[nnodes];

/* Update the number of nodes. */

    nnodes++;

/* Decrease the errors of all nodes */

    for (int i=0; i<nnodes; i++)
    {
       errorsum[i]=errorsum[i]*errordecay;
       signalcounter[i]=signalcounter[i]*errordecay;
    }
    if (nnodes>350)_reset(); // restart if nnodes becomes huge
} // end of add a node section.
```