APPENDIX

```
 program cartogram (output);

 {Pascal translation of Basic V MakeCarto program.}

 {This version is geared to real numbers as the
  mainframe it was tested on appears not to realize
  that life is much easier without them. The Basic
  and C versions which were actually used ran on
  Archimedes and Sun machines with RISC chips in them-
  both were of course much faster (a Fortran translation
  was made - this is possible, but, like most things in
  that language, not a good idea). Pascal is used here
  as it is most likely to be understood.}

 {The two recursive procedures and tree structure are not
  strictly neccessary, but speed things up by a couple
  of orders of magnitude or more, and so are included.}

 {Constants are currently set for the 64 counties
  and 10,000 iterations - a suitably large number
  (Counties do actually converge very quickly - there
  are no problems with the algorithm's speed -
  in fact it appears to move from O(n*n) to O(n log n)
  until other factors come into play when n reaches
  between 10,000 and 100,000 zones...}

 const
   itters = 10000;
   zones = 64;
   ratio = 0.4;        {has to be some-what less than 0.5}
   friction = 0.25;    {this is another magic number - explained elsewhere}
   pi = 3.141592654;

type
   vector = array [1..zones] of real;
   index = array [1..zones] of integer;
   vectors = array [1..zones, 1..21] of real;      {no zone I know of has}
   indexes = array [1..zones, 1..21] of integer;   {more than 21 neighbours}
   leaves = record
              id        : integer;
              xpos      : real;
              ypos      : real;
              left      : integer;
              right     : integer;
   end;
   trees = array [1..zones] of leaves;

var
   infile, outfile           :text;      {input and output files}
   list                      :index;     {list for nearest neighbours}
   tree                      :trees;     {tree structure - see below}
   widest, distance          :real;
```

```
  closest, overlap          :real;
  xrepel, yrepel, xd, yd    :real;       {Suitably small distance units}
  xattract, yattract        :real;       {should be used - for Britain}
  displacement              :real;       {metres is standard. It makes}
  atrdst, repdst            :real;       {little difference if reals are}
  total_distance            :real;       {used, on most machines integers}
  total_radius, scale       :real;       {are much faster and more sensible}
  xtotal, ytotal            :real;       { - even for gravity type models!}
  zone, nb                  :integer;
  other, itter              :integer;
  end_pointer, number       :integer;
  x, y                      :index;     {arrays for zone centroids}
  xvector, yvector          :vector;    {arrays for zone velocities}
  perimeter, people, radius :vector;    {other information about the zones}
  border                    :vectors;   {border lengths between zones}
  nbours                    :index;     {number of neighbours per zone}
  nbour                     :indexes;   {zone neighbours - 0 for the sea}


 {Recursive procedure to add the zone designated by global variable}
 {"zone" to the "tree" structure - this was written in a hurry, is messy}
 {but works - I'm afraid it uses a lot of global variables, but}
 {the structure is probably well known to any reader who already works with}
 {computers and geographic data.}

procedure add_point(pointer,axis :integer);
    begin
      if tree[pointer].id = 0 then      {there is a free leaf so}
        begin                           {put the zone on it}
          tree[pointer].id   := zone;
          tree[pointer].left := 0;
          tree[pointer].right:= 0;
          tree[pointer].xpos := x[zone];
          tree[pointer].ypos := y[zone];
        end
      else                                       {Decide which way to go}
        if axis = 1 then                         {down the tree depending}
          if x[zone] >= tree[pointer].xpos then  {on whether we are at a}
            begin                                {horizontal or vertical}
              if tree[pointer].left = 0 then     {"branch" and where the}
                begin                            {zone to be placed is."}
                  end_pointer := end_pointer +1;
                  tree[pointer].left := end_pointer;
                end;
              add_point(tree[pointer].left,3-axis);
            end
          else
            begin
              if tree[pointer].right = 0 then
                begin
                  end_pointer := end_pointer +1;
                  tree[pointer].right := end_pointer;
                end;
              add_point(tree[pointer].right,3-axis);
```

```
          end
      else
        if y[zone] >= tree[pointer].ypos then
          begin
            if tree[pointer].left = 0 then
              begin
                end_pointer := end_pointer +1;
                tree[pointer].left := end_pointer;
              end;
            add_point(tree[pointer].left,3-axis);
          end
        else
          begin
            if tree[pointer].right = 0 then
              begin
                end_pointer := end_pointer +1;
                tree[pointer].right := end_pointer;
              end;
            add_point(tree[pointer].right,3-axis);
          end
  end;

{This procedure recursively recovers the "list" of zones within}
{"distance" horizontally or vertically of the "zone" from}
{the "tree". The list length is given by "number"}

 procedure get_point(pointer, axis :integer);
   begin
    if pointer>0 then
     if tree[pointer].id > 0 then
      begin
        if axis = 1 then
          begin
            if x[zone]-distance < tree[pointer].xpos then
              get_point(tree[pointer].right,3-axis);
            if x[zone]+distance >= tree[pointer].xpos then
              get_point(tree[pointer].left,3-axis);
          end;
        if axis = 2 then
          begin
            if y[zone]-distance < tree[pointer].ypos then
              get_point(tree[pointer].right,3-axis);
            if y[zone]+distance >= tree[pointer].ypos then
              get_point(tree[pointer].left,3-axis);
          end;
        if (x[zone]-distance < tree[pointer].xpos)
           and (x[zone]+distance >= tree[pointer].xpos) then
          if (y[zone]-distance < tree[pointer].ypos)
             and (y[zone]+distance >= tree[pointer].ypos) then
            begin
              number := number +1;
              list[number] := tree[pointer].id;
            end;
      end;
```

```
    end;

  {Here's the program, first of all set input and output}
  {and intitialize a few things.}

   begin
    reset(infile,'FILE=county.in');
    rewrite(outfile,'FILE=county.out');

    total_distance :=0;
    total_radius := 0;

  {read in the data (an example input file is shown elsewhere) and}
  {find a standard scale for calculating the zone's circle radii.}

    for zone := 1 to zones do
      begin
        read(infile, people[zone],x[zone], y[zone], nbours[zone]);
        perimeter[zone] := 0;
        for nb := 1 to nbours[zone] do
          begin
            read(infile,nbour[zone,nb], border[zone,nb]);
            perimeter[zone] := perimeter[zone] + border[zone,nb];
            if nbour[zone,nb] > 0 then
              if nbour[zone,nb] < zone then
                begin
                  xd := x[zone]- x[nbour[zone,nb]];
                  yd := y[zone]- y[nbour[zone,nb]];
                  total_distance := total_distance + sqrt(xd*xd+yd*yd);
                  total_radius := total_radius + sqrt(people[zone]/pi)
                                  + sqrt(people[nbour[zone,nb]]/pi);
                end;
          end;
        readln(infile);
      end;

   writeln ('Finished reading in topology');

   scale := total_distance / total_radius;
   widest := 0;                            {widest is to be the radius}
                                           {of the widest circle.}
   for zone := 1 to zones do
    begin
      radius[zone] := scale * sqrt(people[zone]/pi);
      if radius[zone] > widest then
        widest := radius[zone];
      xvector[zone] := 0;
      yvector[zone] := 0;
    end;

   writeln ('Finished scaling by ',scale,' widest is ',widest);

{main iteration loop of cartogram algorithm}
```

```
   for itter := 1 to itters do
     begin

{bit of proggy to create a tree}

       for zone := 1 to zones do
         tree[zone].id := 0;
       end_pointer := 1;
       for zone := 1 to zones do
         add_point(1,1);

 {end of esoteric tree building}

       displacement := 0.0;        {to keep a note of how much}
                                   {things are moving.}
{loop of independent displacements}

       for zone := 1 to zones do
         begin
           xrepel  := 0.0;
           yrepel  := 0.0;
           xattract := 0.0;
           yattract := 0.0;
           closest := widest; {to find out the closest neighbour}

 {get all points within widest+radius(zone) into list of length "number"}

           number := 0;
           distance := widest + radius[zone];
           get_point(1,1);

 {work out repelling force of overlapping neighbours}

           if number > 0 then
             for nb := 1 to number do
               begin
                 other := list[nb];
                 if other <> zone then
                   begin
                     xd := x[zone]-x[other];
                     yd := y[zone]-y[other];
                     distance := sqrt(xd * xd + yd * yd);
                     if distance < closest then
                       closest := distance;
                     overlap := radius[zone] + radius[other] - distance;
                     if overlap > 0.0 then
                      if distance > 1.0 then
                       begin
                        xrepel := xrepel - overlap*(x[other]-x[zone])/distance;
                        yrepel := yrepel - overlap*(y[other]-y[zone])/distance;
                       end;
                   end;
               end;
```

```
{work out forces of attraction between neighbours}

          for nb := 1 to nbours[zone] do
            begin
              other := nbour[zone,nb];
              if other <> 0 then
                begin
                  xd := x[zone]-x[other];
                  yd := y[zone]-y[other];
                  distance := sqrt(xd * xd + yd * yd);
                  overlap := distance - radius[zone] - radius[other];
                  if overlap > 0.0 then
                   begin
                    overlap := overlap * border[zone,nb] / perimeter[zone];
                    xattract := xattract + overlap*(x[other]-x[zone])/distance;
                    yattract := yattract + overlap*(y[other]-y[zone])/distance;
                   end;
                end;
            end;

{now work out the combined effect of attraction and repulsion}

          atrdst := sqrt(xattract*xattract+yattract*yattract);
          repdst := sqrt(xrepel*xrepel+yrepel*yrepel);
          if repdst > closest then     {Things are too close, scale them}
            begin                      {down to avoid "whiplash" effects}
              xrepel := closest * xrepel / (repdst + 1);
              yrepel := closest * yrepel / (repdst + 1);
              repdst := closest;
            end;
          if repdst > 0 then
            begin
              xtotal := (1-ratio)*xrepel+ratio*(repdst*xattract/(atrdst+1));
              ytotal := (1-ratio)*yrepel+ratio*(repdst*yattract/(atrdst+1));
            end
          else      {nothing's overlapping}
            begin
              if atrdst > closest then  {don't move too fast!}
                begin
                  xattract := closest*xattract/(atrdst+1);
                  yattract := closest*yattract/(atrdst+1);
                end;
              xtotal := xattract;
              ytotal := yattract;
            end;

{record the vector for posterity}

          xvector[zone] := friction *(xvector[zone]+xtotal);
          yvector[zone] := friction *(yvector[zone]+ytotal);
          displacement := displacement + sqrt(xtotal*xtotal+ytotal*ytotal);
        end;

{update the positions}
```

```
      for zone := 1 to zones do
        begin
          x[zone] := x[zone] + round(xvector[zone]);
          y[zone] := y[zone] + round(yvector[zone]);
        end;
      displacement := displacement / zones;
      writeln('Iteration ', itter, ' displacement ', displacement);
    end;

{we've finished all the iterations so}
{write out the new file}

    for zone := 1 to zones do
        writeln(outfile,radius[zone]:9:0,',',x[zone]:9,',',y[zone]:9);
  end.
```