A Quadrilateral Approach to Congressional Districting

February 12, 2006

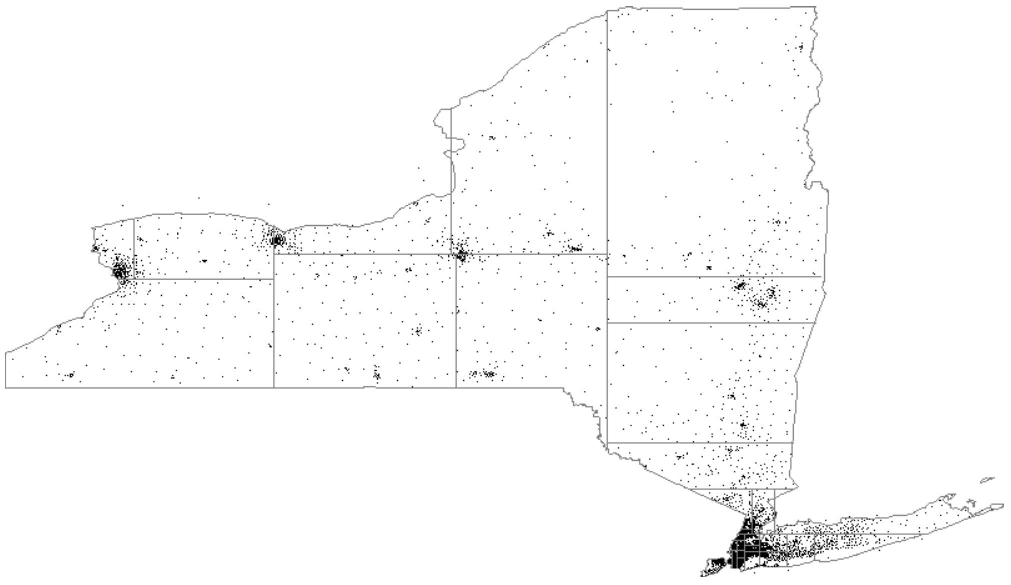
Abstract

Congressional districting has long been a problem in the United States; boundaries are manipulated for political gain by whomever is in power. To correct this requires a method of producing districts without any human input; by eliminating any opportunity for tampering with the process, it is hoped that trust can be restored in this part of the political process.

We present an algorithm that is deterministic, requiring only a map and population distribution of a state, producing a clear layout of the congressional districts.

This algorithm was designed primarily for simplicity; our goal was to produce results that could be checked by anyone, thus ensuring that the algorithm was followed and not modified for political gain. Not only should the algorithm be simple, but also the resulting districts; boundaries should be as clear as possible, to eliminate potential confusion and opportunistic election tampering.

To this end, our algorithm produces rectangular districts that are maximally compact given the constraints in shape and population distribution. We used only U.S. Census data sets as input for our algorithm, mimicking actual use. This algorithm was tested on a number of states to ensure that it produced reasonable results, and we believe we have achieved our goal.



Contents

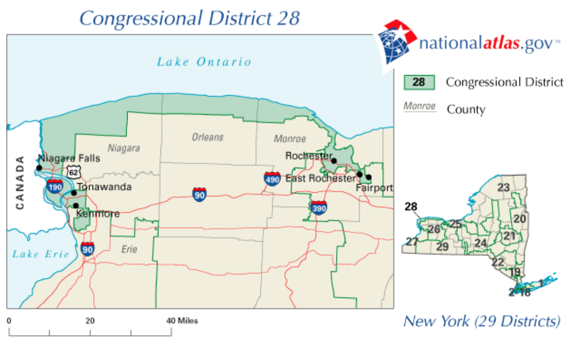


Figure 1: New York's 28th Congressional District.

[4] [1] [2]

1 Introduction

Elbridge Gerry is best known for the term coined in his name — Gerrymandering.[3] The practice certainly goes back further in time, but never had it been so obvious. The redistricting during his 1810-1812 term cost him his position as Governor and presented a problem with Congressional districting in the United States. Though obvious to political observers, the practice still manages to slip through the cracks in the political system. *Baker v. Carr* marked a historical turning point in the way congressional districts are established. The court ruled that many districts were in violation of the Fourteenth Amendment and required reapportioning. The question remains, however, of how to draw the districts of a state from a neutral standpoint with no influences from the race, gender, party affiliation or otherwise.

For our models we have used the Census Tract population and boundary data from the year 2000. The precision of data is accurate in most places, but there are obvious faults when the population and boundary values are plotted by latitude and longitude. There are instances where population points fall outside of the boundary defined for the state.

2 A Baseline Approach

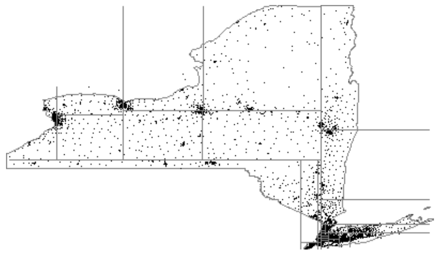


Figure 2: Algorithm applied to New York.

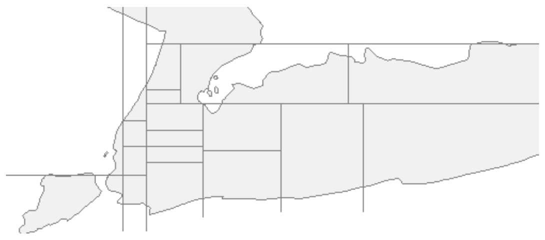


Figure 3: New York City

2.1 Overview

This method uses only census tracts to determine rectangular districts of roughly equal population. Under this algorithm, a state is divided repeatedly into smaller regions until each region represents only one district.

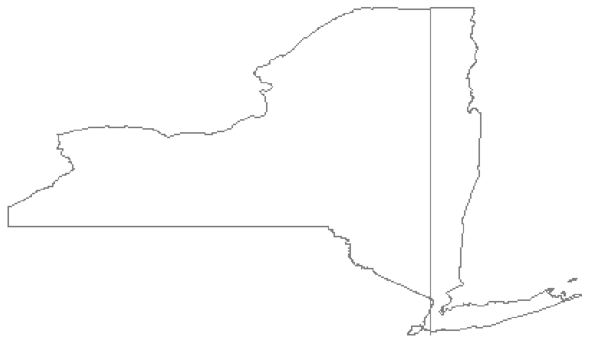
2.2 Design

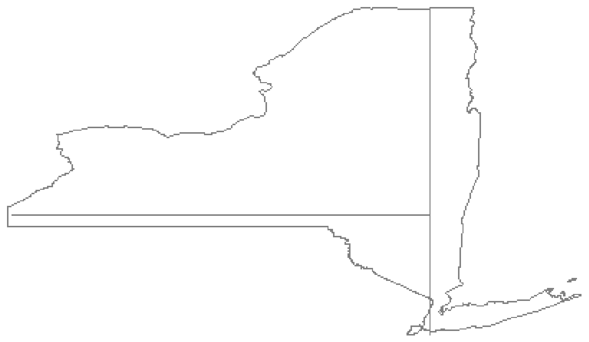
```
Algorithm 2.1: DIVIDEDISTRICT($district, districtSize$)

if $districtSize = 1$
  then {$break$
if $district \rightarrow width > district \rightarrow height$
  then {$drawLongitudinalLine(districtSize)$
  else {$drawLatitudinalLine(districtSize)$

$divideDistrict(largeSide, \lceil districtSize/2 \rceil)$
$divideDistrict(smallSide, \lfloor districtSize/2 \rfloor)$
```

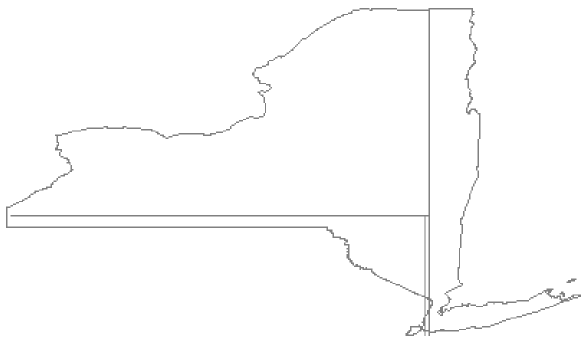
For each recursion, a bounding rectangle is constructed around the data points the recursion has been tasked to divide. Bounding coordinates are examined to determine the direction to draw the line. If the bounding rectangle has a greater span of longitudinal distance, a line of latitude is drawn, else a line of longitude is drawn. If the recursion has been tasked with dividing an even number of districts, these districts are split in half. If an odd number must be split, the algorithm will calculate the population relative to the parallel computed by the previous function call. This comparison determines which side to pass the greater number of districts to.

Next, the data points are sorted and a line is found that approximately satisfies the ratios. The longitudinal range of the rectangle is converted to a distance at the latitude of the midpoint. If this distance is greater than the span of the box's latitude, a line of longitude is drawn, otherwise it draws a line of latitude. This process attempts to minimize perimeter by drawing the shortest line through the box while preserving the ratio. By examining the physical dimensions of the state, it progresses by comparing the total width and height measured from the extrema.

A visualization of the first iterations of the algorithm on the state of New York:


(a) Iteration 1


(b) Iteration 2


(c) Iteration 3

2.3 Analysis

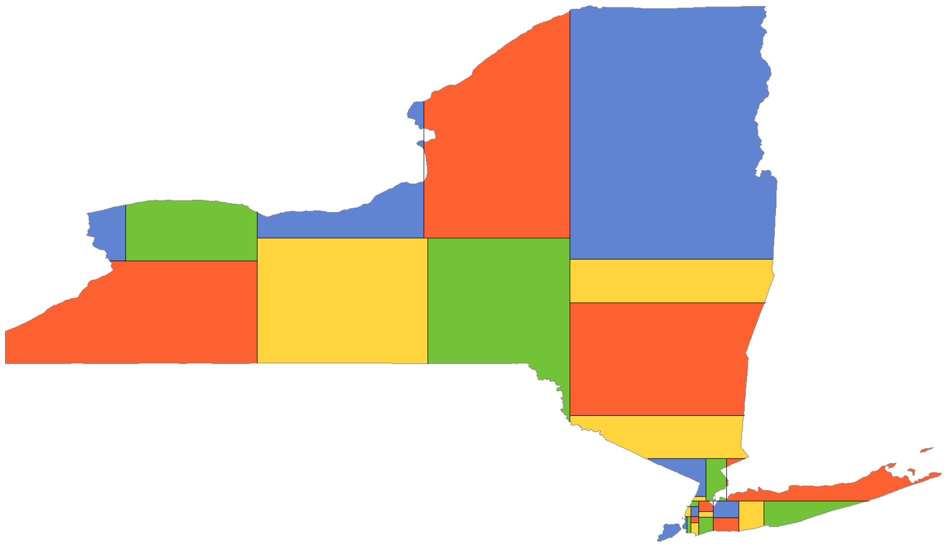
The primary consideration in the selection of the algorithm was that it be entirely deterministic; by removing any human element from the districting process, all opportunity for gerrymandering is removed.

This has some consequences; most notably that it is not practical to take geography into account, such as rivers and mountains, that might provide natural boundaries. Using natural geography as borders would require a person to determine prior to running the algorithm which geography makes suitable borders; as there is no truly objective way of making this determination, and it would depend on the data (topography maps) being used, it is not an option in an algorithm designed to be deterministic.

Another consideration is that the generated districts be simple. While it would not be difficult to produce an algorithm to generate districts bounded by arbitrary straight lines, we decided to only use north-south and east-west lines. This produces simpler shapes, defined by latitude and longitude lines, meaning that it is easier for an average person to see how a district is defined. More importantly, the algorithm can be simpler, and an average person can verify that it was followed, and be more confident in the redistricting process.

For New York State, the problem caused by sectioning with regard only to population density, without border data manifests early on. The first divide occurs longitudinally on the eastern side of the state splitting New York City in two. The algorithm then begins to the west of the divide, realizes that this section is wider latitudinally and thus wants to cut longitudinally. The extreme population of the New York City area causes it to make the division very near the last division, however, resulting in a thin slice. The slice is broken up into further pieces, this time cutting horizontally. Due to the fact that the southern half of this division is so heavily populated, it draws the next line in the middle of the city. This result is fine for the sections near New York City, but leaves a large and awkward remaining slice directly northwards.

3 The Most Compact Rectangular District Algorithm



3.1 Overview

This algorithm is recursive: given some area, it finds the shortest line dividing a region into areas of equal population, or nearly equal if the area was to be divided into an odd number of districts, then recurses on each of the new areas.

This has the effect of creating districts which are maximally compact given the constraints; that is, rectangular and containing equal population. The algorithm avoids creating long, narrow districts in favor of districts that are more square. Our implementation of this algorithm also takes state borders into account, correctly handling cases like the peninsula of New York in finding shortest lines.

3.2 Design

```
Algorithm 3.1: DIVIDEDISTRICT($district, districtSize$)

  if $districtSize = 1$
    then {$break$
  $line = findShortestLine(district, districtSize)$
  $drawLine(line)$
  $divideDistrict(largeSide, \lceil districtSize/2 \rceil)$
  $divideDistrict(smallSide, \lfloor districtSize/2 \rfloor)$
```

```
Algorithm 3.2: FINDSHORTESTLINE($district, districtSize$)

$dir_i[4] = [N, S, E, W]$;
$targetPop = \lfloor districtSize/2 \rfloor$;
$bestLine$;
comment: Line to return
for $i = 0$ to $3$
    $newLine = calcLine(district, dir_i, targetPop)$
    comment: Calculates the distance of a partition line
    if $newLine < bestLine$
        then $bestLine = newLine$
return ($bestLine$)
```

Unlike the base algorithm, this method computes the direction of the line to draw in addition to the ratio and boundary distance.

3.3 Analysis

There are some properties of our algorithm that may be perceived as disadvantages. In general, it will tend to split up areas of high population density and group them with areas of low population density, to a certain extent; the degree to with this actually happens depends very much on the state. For example, were Alaska to be given two districts with our algorithm, Anchorage would be divided exactly in half, and each district would be divided approximately evenly between urban and rural populations. Were a human to divide Alaska into two districts, he or she might choose to have one of them contain only Anchorage and the rest containing the rest of the state, thereby uniting areas with similar demographics in the same district. However, for the states we have run our algorithm on this does not appear to happen very much in practice. The results appear to be particularly good for New York.

The algorithmic efficiency is linear here as well, determined only by the number of districts for the state.

The results are quite accurate, and could be made more so with better data. Using the equation for standard deviation,

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=0}^{N} x_i - \bar{x}} \tag{1}$$

we determine that the district populations do not stray far from the mean. For New York, $\sigma = 3879.51$. The mean, standard deviation, and representatives per state for each state with more than one representative can be found in Appendix 1.

4 Further Possibilities

4.1 The Error Minimization Algorithm

One of the major issues with Congressional Districting is dividing the population evenly. The data we have used for these models is rather accurate, but the algorithm will produce better results with better data. With this data, a possible method of minimizing the error is by choosing sections with only regard to how close the ratio of the population is to the desired ratio.

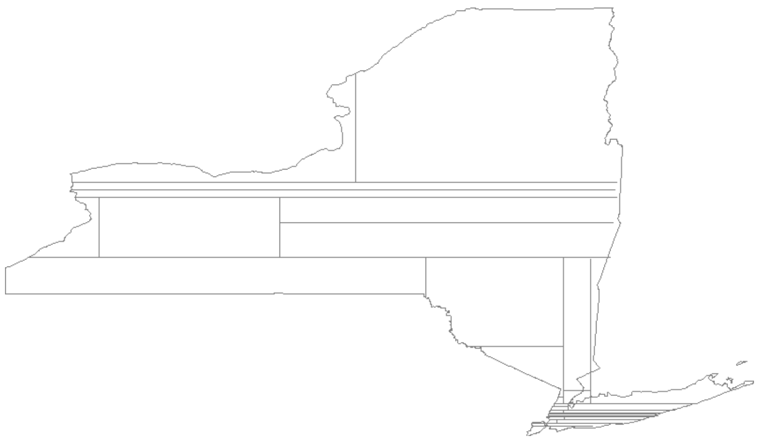


Figure 4: Application of the Error Minimization Algorithm

After application of this algorithm the flaws are readily evident, as seen in Figure 4. The shape of the districts produced are simple, but hardly desirable. The algorithm does, however, produce districts with a small margin of error. The range of the districts is less than five thousand as compared to over twelve thousand in the MCRD algorithm, while the standard deviation is less than a third of MCRD.

Overall, this algorithm is not acceptable due to the shapes of the districts it creates. Taking error into consideration in other methods is reasonable, but should not be a sole deciding factor.

4.2 A Brute Force Approach to the Problem

A second possibility is the brute force approach to the problem. Following closely to the Most Compact Rectangular District algorithm, this method could look ahead through the recursive trees produced by the four different ways to draw lines as defined in MCRD. Though horribly inefficient, it could be viable for states with less districts. On a modern machine, our implementation of the MCRD algorithm can process any state in less than two and a half seconds. If parallelized and optimized, this technique could be plausible. Checking each and every possibility would be a way of making sure to get a maximized result.

5 Conclusions

Our algorithm shows that it is possible to deterministically divide states into districts, eliminating the need for human involvement.

Our algorithm produces simple rectangular districts, defined only by lines of latitude and longitude, without regard to party affiliation, directly or indirectly.

It is our hope that use of our algorithm in redistricting will lead to more competitive elections and a fairer democracy.

References

[1] Census 2000: State and state equivalent areas cartographic boundary files. Census 2000 Boundary Files, 2000.

[2] Census 2000 u.s. gazetteer files. Census 2000 Tract Data, 2000.

[3] Gary W. Cox and Jonathan N. Katz. *Elbridge Gerry's Salamander: The Electoral Consequences of the Reapportionment Revolution.* Cambridge University Press, 2002.

[4] Mark Monmonier. *Bushmanders and Bullwinkles.* The University of Chicago Press, 2001.

6 Appendix

Each of the tables and figures in this section apply towards the MCRD Algorithm.

6.1 Mean/Standard Deviation of District Population by State

State	μ	σ	N	*State*	μ	σ	N
AL	635300.00	2549.46	7	MS	711164.50	4142.39	4
AZ	641329.00	6091.77	8	MO	621690.11	5302.29	9
AR	668350.00	1866.07	4	NE	570421.00	3370.07	3
CA	639087.70	5113.10	53	NV	666085.67	2179.20	3
CO	614465.86	4251.38	7	NH	617893.00	290.00	2
CT	681113.00	5399.40	5	NJ	647257.69	3389.83	13
FL	639295.12	4480.11	25	NM	606348.67	1820.44	3
GA	629727.15	4006.71	13	NY	654360.59	3879.51	29
HI	605768.50	940.50	2	NC	619177.92	5008.13	13
ID	646976.50	3447.50	2	OH	630730.00	3829.10	18
IL	653647.00	4662.14	19	OK	690130.80	4418.53	5
IN	675609.44	4304.317	9	OR	684279.80	4615.70	5
IA	585264.80	3328.49	5	PA	646371.26	2874.79	19
KS	672104.50	2345.95	4	RI	524159.50	1470.50	2
KY	673628.17	4190.65	6	SC	668668.67	4232.97	6
LA	638425.14	4919.99	7	TN	632142.56	3054.41	9
ME	637461.50	0.50	2	TX	651619.38	5129.03	32
MD	662060.75	4331.22	8	UT	744389.67	4935.44	3
MA	634909.70	4884.10	10	VA	643501.36	3455.79	11
MI	662562.94	2484.62	15	WA	654902.33	4008.80	9
MN	614934.88	2615.11	8	WV	602781.33	1023.89	3
				WI	670459.38	2472.80	8

6.2 Districts vs. Population

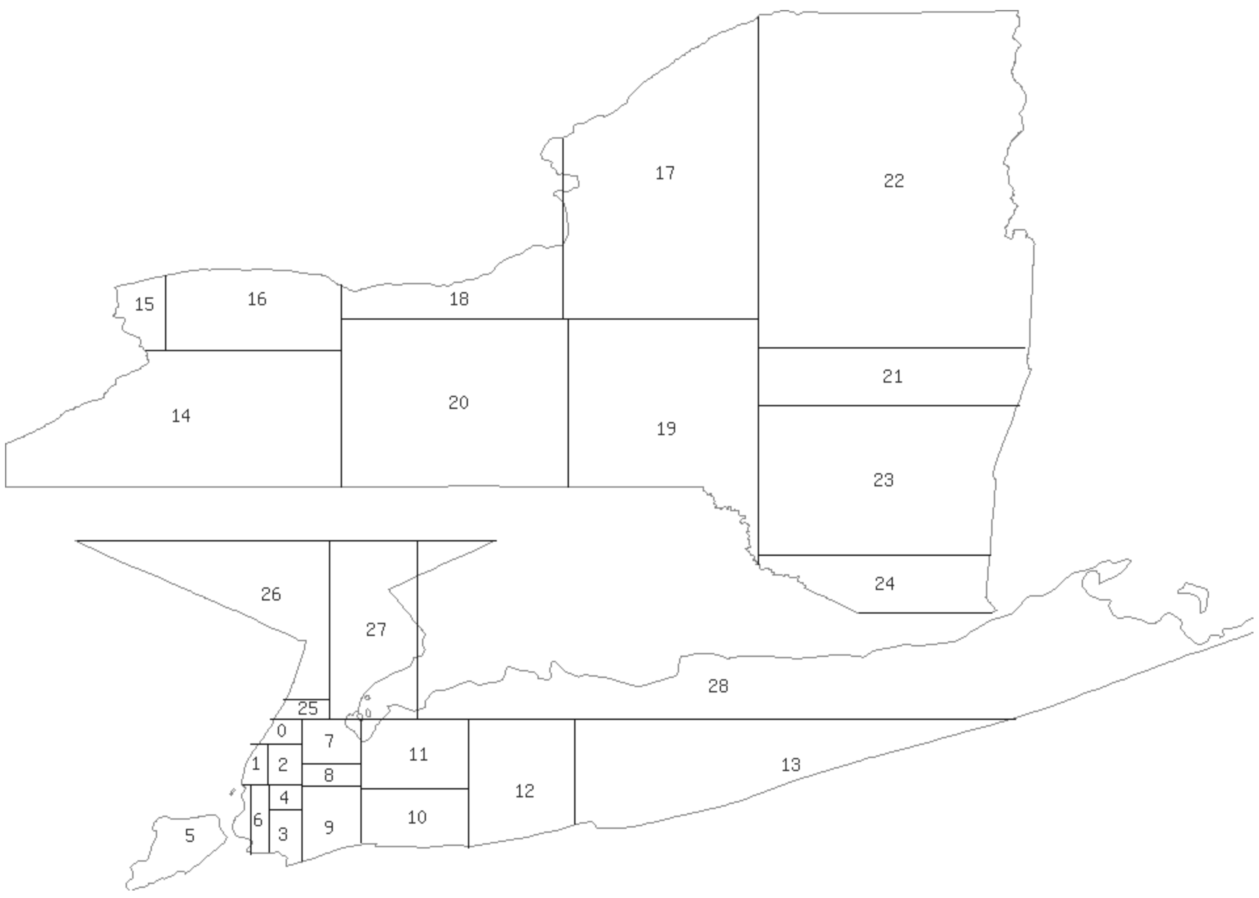


Figure 5: Congressional Districts for New York by number, NYC enhanced for detail.

District	*Population*	*District*	*Population*
0	658898	15	659152
1	656267	16	652944
2	653822	17	657645
3	657473	18	652706
4	650712	19	653776
5	658679	20	648462
6	648803	21	658919
7	659212	22	651779
8	652338	23	657136
9	650157	24	649878
10	657118	25	659973
11	652300	26	649526
12	655847	27	659230
13	650230	28	647173
14	656232		

6.3 Code

```
#!/usr/bin/env perl
use strict;
use warnings;

use List::Util qw/min max/;
use Geo::ShapeFile;

die "Usage: $0 db_file state districts width out_file\n" unless @ARGV >= 3;
my ($db_file, $state, $total_districts, $width, $out_file) = @ARGV;
my @district_pops;
my $line_number = 0;

# The overall map coordinates for converting pixels to points
my $map = {
    width    => $width,
    height   => 0, # generate from width and max coords
    max_lat  => -181,
    max_lon  => -181,
    min_lat  =>  181,
    min_lon  =>  181,
};
my $bounds = get_bounds($state); # boundaries of the state

# Get maximum and minimum coordinates for the map
{
    for my $poly (@$bounds) {
        for (@$poly) {
            $map->{max_lon} = $_->{x} if $_->{x} > $map->{max_lon};
            $map->{min_lon} = $_->{x} if $_->{x} < $map->{min_lon};
            $map->{max_lat} = $_->{y} if $_->{y} > $map->{max_lat};
            $map->{min_lat} = $_->{y} if $_->{y} < $map->{min_lat};
        }
    }
};

my $height = $width *
    ($map->{max_lat} - $map->{min_lat}) /
    ($map->{max_lon} - $map->{min_lon});
$map->{height} = $height;

use GD;
my $im = GD::Image->new($width, $height);
my $white = $im->colorAllocate(255, 255, 255);
my $district_number = 0;

my $total_pop = 0; # population of the whole state
```

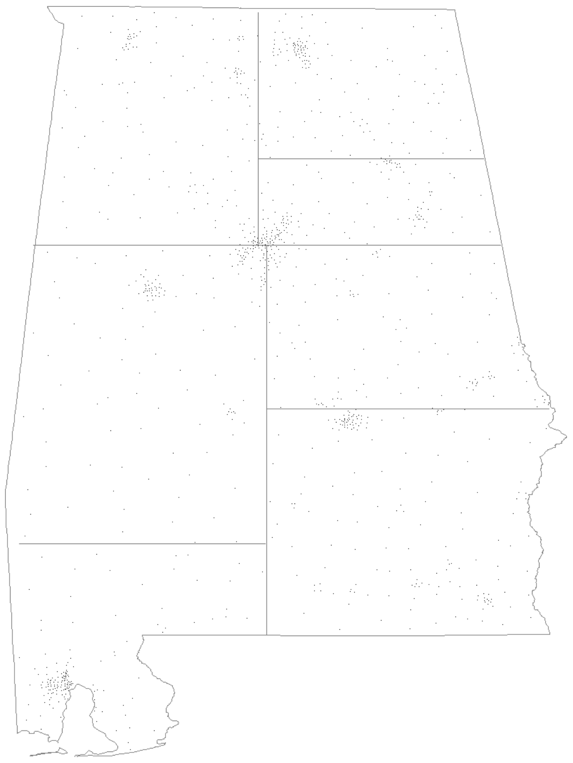
```
{

    open my $fh, "<", $db_file or die "$db_file: $!";
    my $data = [];
    while (<$fh>) {
        next if !/^\Q$state\E/io; # fetch all records for a particular state
        my @cols = split " "; # grab whitespace-separated columns
        push @{$data}, {
            pop => $cols[1],
            lat => $cols[7],
            lon => $cols[8],
        };
        $total_pop += $cols[1];
    }
    close $fh;

    $map->{span_lat} = abs $map->{max_lat} - $map->{min_lat};
    $map->{span_lon} = abs $map->{max_lon} - $map->{min_lon};

    # Draw on boundaries
    my $bound_color = $im->colorAllocate(128, 128, 128);
    for my $poly (@$bounds) {
        for (my $i = 0; $i < $#$poly; $i++) {
            $im->line(
                pt_to_px($map, $poly->[$i]{x}, $poly->[$i]{y}),
                pt_to_px($map, $poly->[$i + 1]{x}, $poly->[$i + 1]{y}),
                $bound_color
            );
        }
    }

    # Draw on some points
    my $point_color = $im->colorAllocate(0, 0, 0);
    for my $d (@$data) {
        my ($x, $y) = pt_to_px($map, $d->{lon}, $d->{lat});
        $im->setPixel($x, $y, $point_color);
    }

    divide_district($total_districts, $data, $total_pop);
    print "Total population: $total_pop\n";
    # Calculate mean and standard deviation
    my $sum = 0;
    $sum += $_ for @district_pops;
    my $q = 0;
    $q += ($_ - $sum / @district_pops) ** 2 for @district_pops;
    print "Mean: ", $sum / @district_pops, "\n";
    print "Standard deviation: ", sqrt $q / @district_pops, "\n";
```

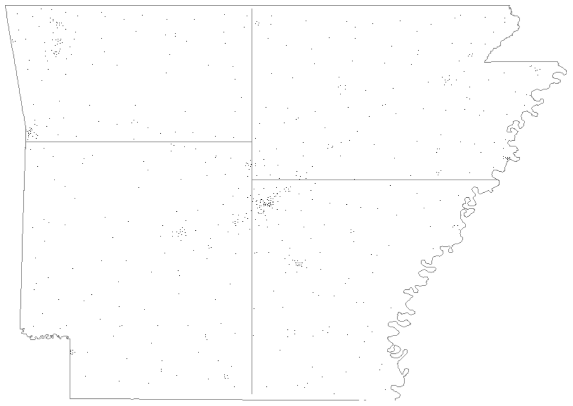
```
        print "Minimum: ", min(@district_pops), "\n";
        print "Maximum: ", max(@district_pops), "\n";
    };

    make_png($out_file);
    sub make_png {
        my $png_file = shift;
        open my $fh, ">", $png_file or die "$png_file: $!";
        print $fh $im->png;
        close $fh;
    }

    sub divide_district {
        my $districts = shift;
        my $data = shift;
        # Pass population from previous iteration to save computation
        my $last_pop = shift;

        my %max = (
            lat => -180,
            lon => -180,
        ); # maximum lat/lon
        my %min = (
            lat => 180,
            lon => 180,
        ); # minimum lat/lon

        for my $d (@$data) {
            $max{lat} = $d->{lat} if $d->{lat} > $max{lat};
            $max{lon} = $d->{lon} if $d->{lon} > $max{lon};
            $min{lat} = $d->{lat} if $d->{lat} < $min{lat};
            $min{lon} = $d->{lon} if $d->{lon} < $min{lon};
        }

        if ($districts == 1) {
            my $label_color = $im->colorAllocate(128, 0, 0);
            # Population in the center of each district
            my @label_coords = pt_to_px($map,
                ($max{lon} + $min{lon}) / 2,
                ($max{lat} + $min{lat}) / 2
            );
            # Population of a single district
            print "Population #$district_number: $last_pop\n";
            push @district_pops, $last_pop;
            return;
        }
        # Number of districts for each side of the line
        my $small_districts = int $districts / 2;
        my $big_districts = $districts - $small_districts;

        # Target population to shoot for the smaller number of districts
        my $target_pop = int $last_pop * $small_districts / $districts;

        # Find positions for lines for each direction
        my %lines = map {
            my ($pos, $pop) = find_line($target_pop, $data, $_);
            warn "\n\n-1!\n\n" if $pos == -1;
            # Minimum coord, maximum coord
            my ($min, $max, $km);
            # and the kilometers between
            if ($_ eq "N" or $_ eq "S") {
                # Check the horizontal distance between the borders
                ($min, $max) = intersection(
                    { x => 0, y => $pos }, $bounds, $min{lon}, $max{lon}, "x"
                );
                # 1 deg lon = 111 km * cos(lat)
            #   ($min, $max) = ($min{lon}, $max{lon});
                $km = 111 * ($max - $min) * cos $pos / 180 * 3.14159;
            }
            else {
                # Check the vertical distance between the borders
                ($min, $max) = intersection(
                    { x => $pos, y => 0 }, $bounds, $min{lat}, $max{lat}, "y"
                );
            #   ($min, $max) = ($min{lat}, $max{lat});
                # 1 deg lat = 111 km
                $km = 111 * ($max - $min);
            }
            $_ => {
                pos => $pos,
                pop => $pop,
                min => $min,
                max => $max,
                km => $km
            };
        } qw/N S W E/;

        # Start from the side with the shortest distance between borders
        my ($start_from) =
            sort {
                $lines{$a}{km} <=> $lines{$b}{km}
            #    abs($target_pop - $lines{$a}{pop}) <=>
            #    abs($target_pop - $lines{$b}{pop})
            } keys %lines;

        my ($x1, $y1, $x2, $y2); # line coordinates
        if ($start_from eq "N" or $start_from eq "S") {
            # Steps along latitude, draw a line of longitude
            ($x1, $y1) = pt_to_px($map,
                $lines{$start_from}{max},
                $lines{$start_from}{pos}
            );
            ($x2, $y2) = pt_to_px($map,
                $lines{$start_from}{min},
                $lines{$start_from}{pos}
            );
        }
        else {
            # Steps along longitude, draw a line of latitude
            ($x1, $y1) = pt_to_px($map,
                $lines{$start_from}{pos},
                $lines{$start_from}{max}
            );
            ($x2, $y2) = pt_to_px($map,
                $lines{$start_from}{pos},
                $lines{$start_from}{min}
            );
        }

        # print "#$line_number: $lines{$start_from}{pos} $start_from\n";

        my $line_color = $im->colorAllocate(128, 128, 128);
        $im->line($x1, $y1, $x2, $y2, $line_color);
        $line_number++;

        my ($small_data, $big_data) = divide_data(
            $data, $lines{$start_from}{pos}, $start_from
        );
        divide_district(
            $small_districts,
            $small_data,
            $lines{$start_from}{pop},
        );
        divide_district(
            $big_districts,
            $big_data,
            $last_pop - $lines{$start_from}{pop},
        );
    }
```

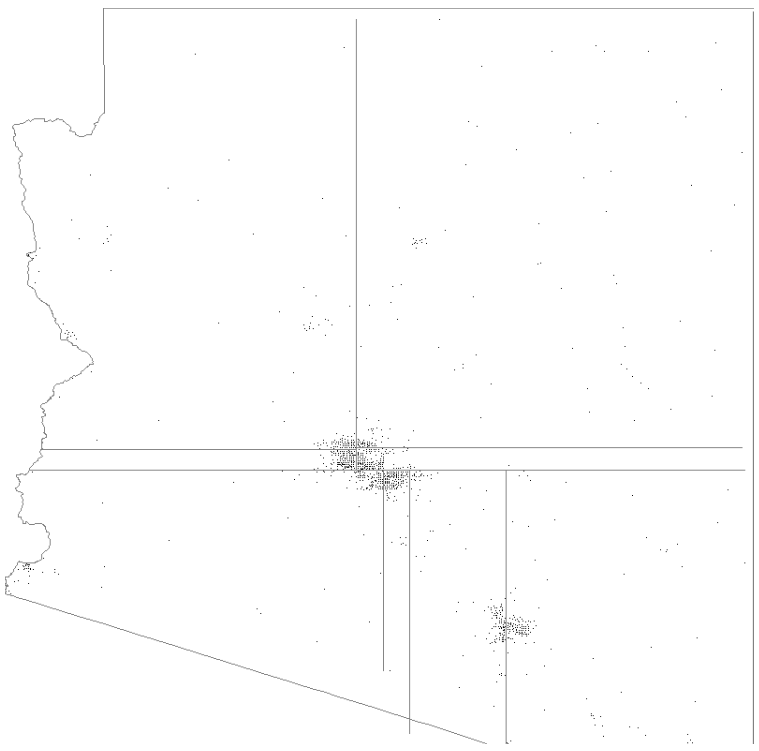
```
# Split data set by a line
sub divide_data {
    my $data = shift;
    my $line_pos = shift;
    my $dir = shift;
    my $first_side = [];
    my $second_side = [];
    for my $d (@$data) {
        if ($dir eq "W") {
            if ($d->{lon} < $line_pos) {
                push @{$first_side}, $d;
            }
            else {
                push @{$second_side}, $d;
            }
        }
        elsif ($dir eq "E") {
            if ($d->{lon} > $line_pos) {
                push @{$first_side}, $d;
            }
            else {
                push @{$second_side}, $d;
            }
        }
        elsif ($dir eq "N") {
            if ($d->{lat} > $line_pos) {
                push @{$first_side}, $d;
            }
            else {
                push @{$second_side}, $d;
            }
        }
        elsif ($dir eq "S") {
            if ($d->{lat} < $line_pos) {
                push @{$first_side}, $d;
            }
            else {
                push @{$second_side}, $d;
            }
        }
    }
    return $first_side, $second_side;
}

sub intersection {
    my $check = shift;
    my $bounds = shift;
    my $min_ = shift;
    my $max_ = shift;
    my $x = shift; # check the ($x eq x ? horizontal : vertical)

    my $min = 100000;
    my $max = -100000;

    my $y = "y";
    $y = "x" if $x eq "y";

    for my $poly (@$bounds) {
        $poly->[@$poly] = $poly->[0];
        for (my $i = 0; $i < $#$poly; $i++) {
            if ($check->{$y} > $poly->[$i]{$y}
                and $check->{$y} > $poly->[$i+1]{$y}) { next; }
            if ($check->{$y} < $poly->[$i]{$y}
                and $check->{$y} < $poly->[$i+1]{$y}) { next; }
    if ($poly->[$i]{$y} == $poly->[$i+1]{$y}) { next; }
#print "checking $poly->[$i]{$y} == $poly->[$i+1]{$y}\n";
#print "$poly->[$i]{$y}    $check->{$y}    $poly->[$i+1]{$y}\n";

            my $weight = ($check->{$y} - $poly->[$i+1]{$y}) /
                ($poly->[$i]{$y} - $poly->[$i+1]{$y});
#print "w=$weight\n";
            my $intersect = $poly->[$i]{$x} + $weight * ($poly->[$i]{$x} - $po
            if ($intersect < $min) { $min = $intersect }
            if ($intersect > $max) { $max = $intersect }
        }
    }

    return max($min, $min_), min($max, $max_);
}

# Whether a point is in a polygon (with counter clockwise coordinates)
sub is_in_poly {
    my $check = shift;
    my $poly = shift;
    $poly->[@$poly] = $poly->[0];
    for (my $i = 0; $i < @$poly; $i++) {
        my $check_trans;
        $check_trans->{x} = $check->{x} - $poly->[$i]{x};
        $check_trans->{y} = $check->{y} - $poly->[$i]{y};

        my $point_trans;
        $point_trans->{x} = $poly->[$i]{x} - $poly->[$i-1]{x};
        $point_trans->{y} = $poly->[$i]{y} - $poly->[$i-1]{y};
```

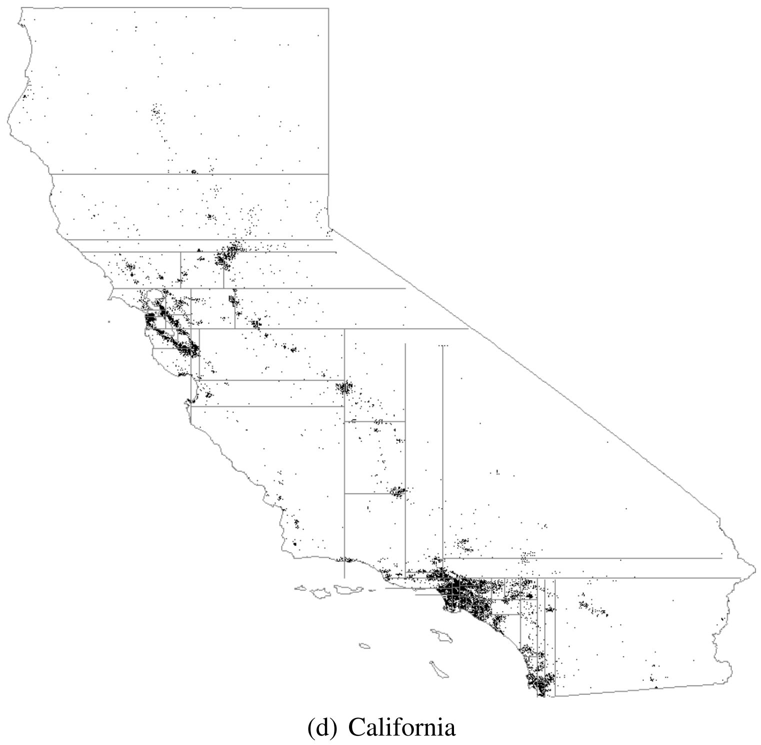
```
            my $point_rot;
            $point_rot->{x} = -$point_trans->{y};
            $point_rot->{y} = $point_trans->{x};

            my $dot = $check_trans->{x} * $point_rot->{x}
                + $check_trans->{y} * $point_rot->{y};
            return 0 if $dot < 0;
        }
        return 1;
    }

    sub pt_to_px {
        my ($map, $x_pt, $y_pt) = @_;
        my $x = int $map->{width} * ($map->{min_lon} - $x_pt) /
            ($map->{min_lon} - $map->{max_lon});
        my $y = int $map->{height} * (1 - (($y_pt - $map->{min_lat}) /
            ($map->{max_lat} - $map->{min_lat})));
        return $x, $y;
    }

    sub get_bounds {
        my $state = shift;
        my $g = Geo::ShapeFile->new("data/shp/$state");
        my @poly;
        for (1 .. $g->shapes) {
            my $r = $g->get_shp_record($_);
            push @poly, [
                map +{ x => $_->X, y => $_->Y }, @{$r->{shp_points}}
            ];
        }
        return \@poly;
    }

    sub find_pop {
        my $data = shift;
        my $min_lat = shift;
        my $min_lon = shift;
        my $max_lat = shift;
        my $max_lon = shift;
        my $pop = 0;
        foreach my $d (@$data) {
            if ($d->{lat} >= $min_lat and
                $d->{lat} <= $max_lat and
                $d->{lon} >= $min_lon and
                $d->{lon} <= $max_lon) {
                $pop += $d->{pop};
            }
        }
        return $pop;
    }

    sub find_line {
        my $target_pop = shift;
        my $data = shift;
        my $dir = shift;
        my $pop = 0;
        my @sorted;

        warn "0!\n" if $target_pop == 0;
        return 0, 0 if $target_pop == 0;

        if($dir eq "S" or $dir eq "N") {
            @sorted = sort { $a->{lat} <=> $b->{lat} } @$data;
        }
        else {
            @sorted = sort { $a->{lon} <=> $b->{lon} } @$data;
        }
        if($dir eq "N" or $dir eq "E") {
            @sorted = reverse(@sorted);
        }

        my $i = 0;
        foreach my $d (@sorted) {
            ++$i;
            $pop+=$d->{pop};
            if($pop >= $target_pop) {
                if ($dir eq "N" or $dir eq "S") {
                    # Return if the next latitude is not equal
                    if (not $sorted[$i] or $d->{lat} != $sorted[$i]{lat}) {
                        return $d->{lat}, $pop;
                    }
                }
                else {
                    if (not $sorted[$i] or $d->{lon} != $sorted[$i]{lon}) {
                        return $d->{lon}, $pop;
                    }
                }
            }
        }
        return '-1', $pop;
    }
```

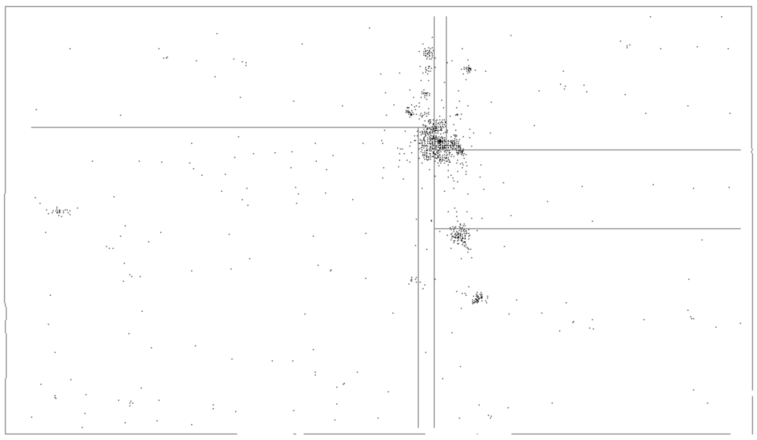
6.4 Algorithm Applied on Each State

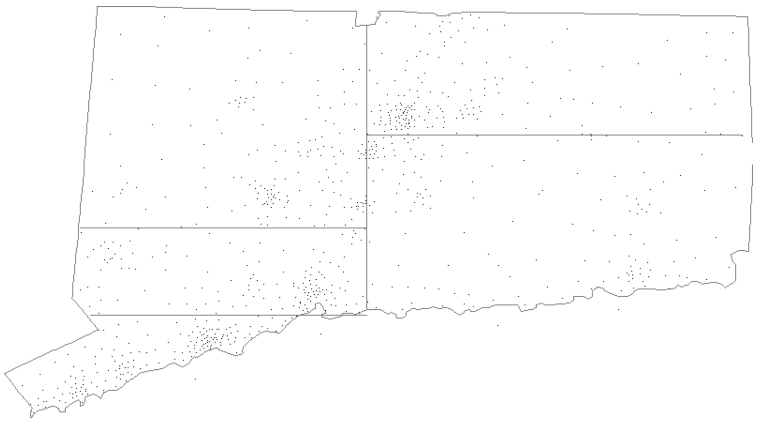

(a) Alabama

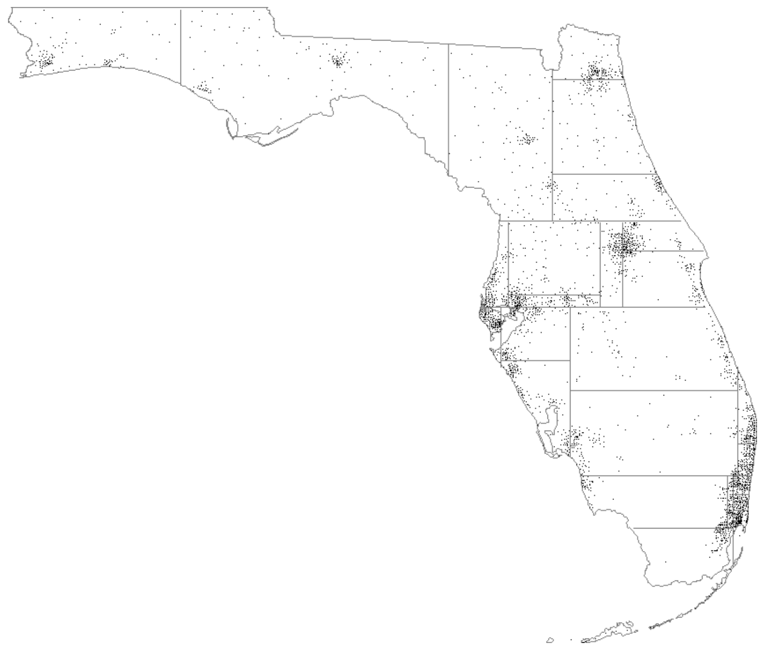

(b) Arkansas


(c) Arizona

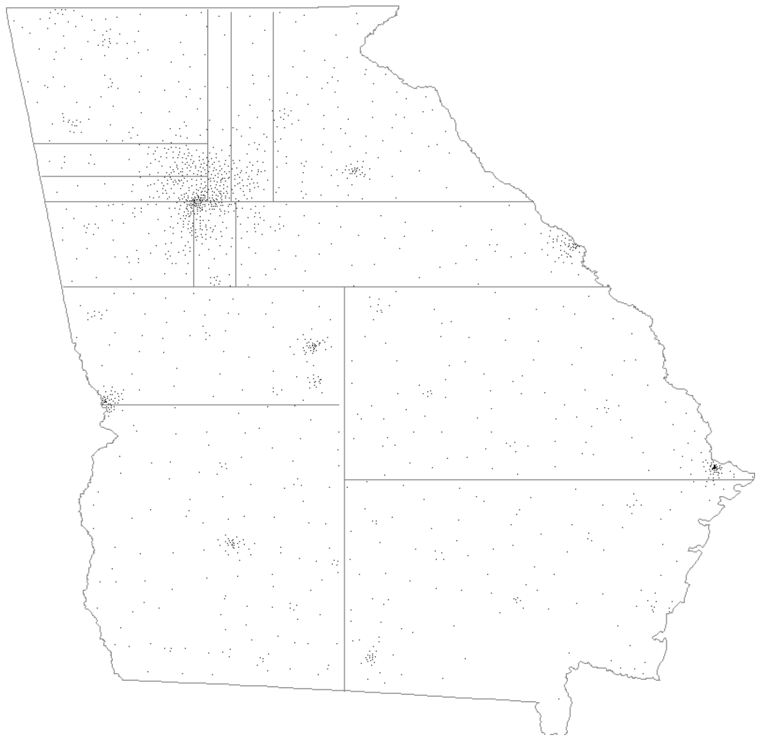

(d) California


(e) Colorado


(f) Conneticut

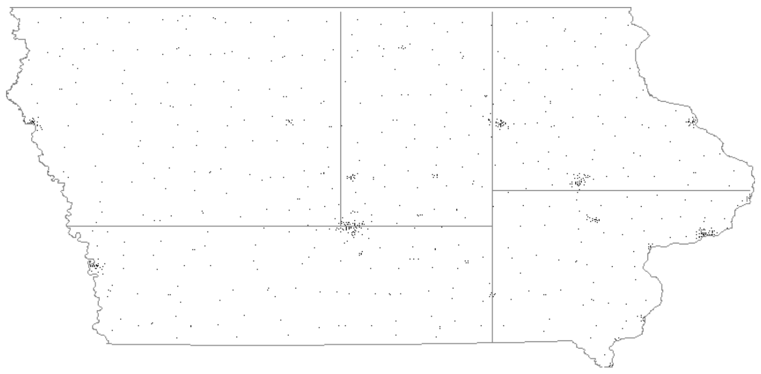


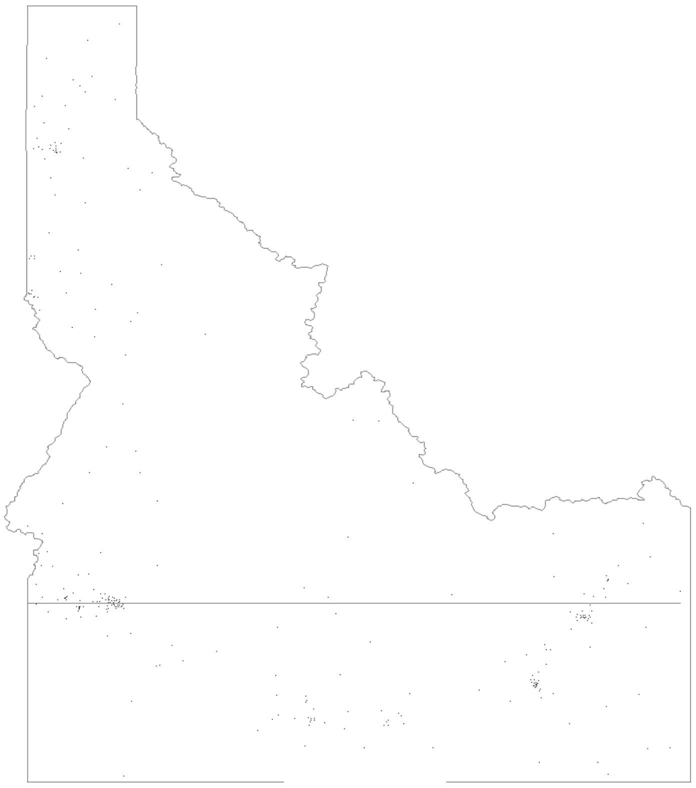
(g) Florida

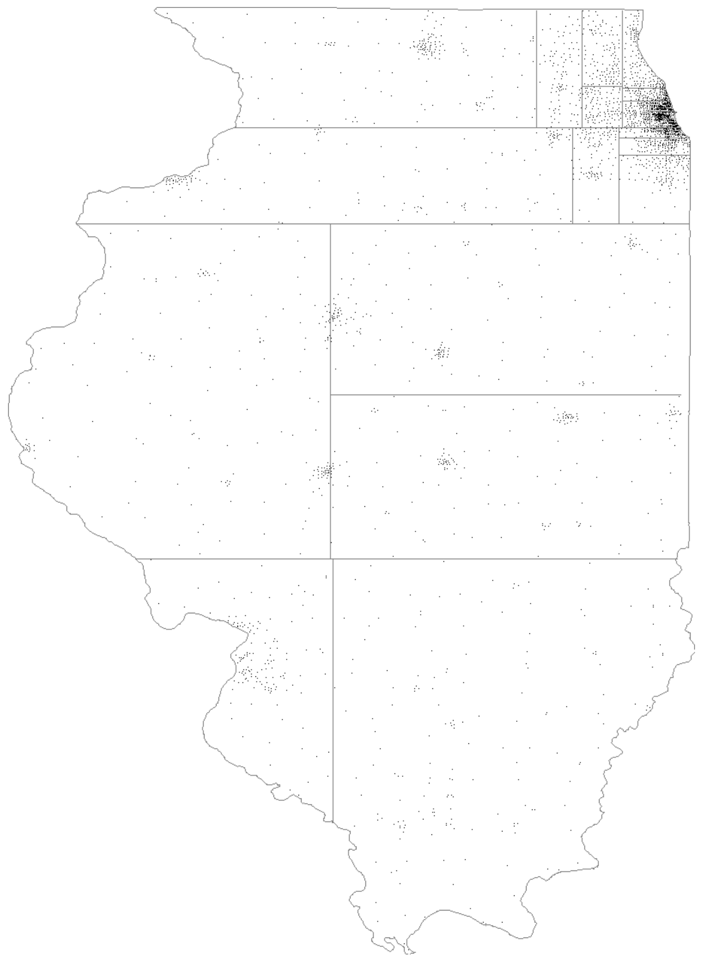


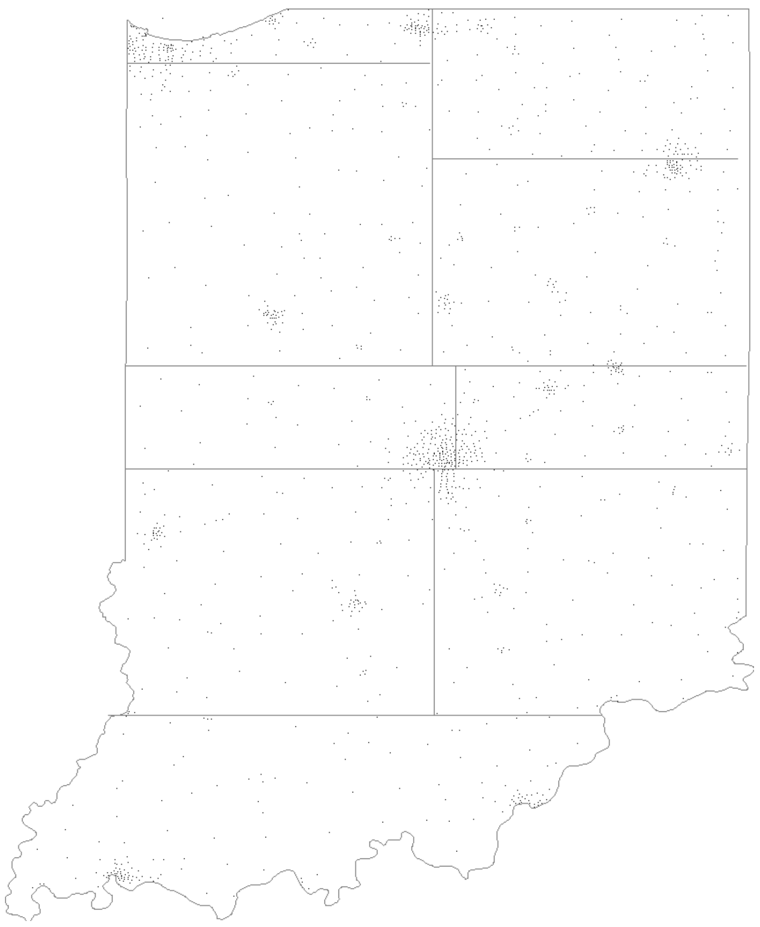
(h) Georgia

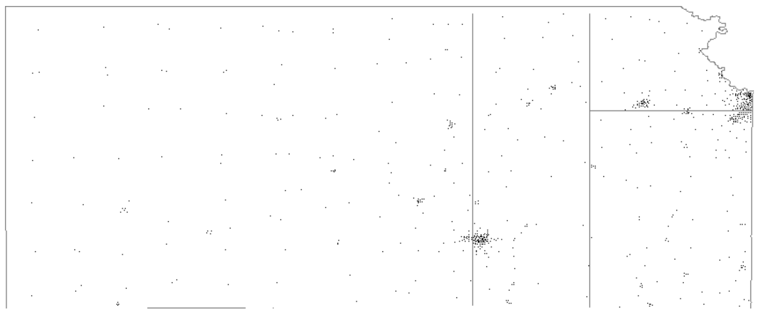
(i) Hawaii

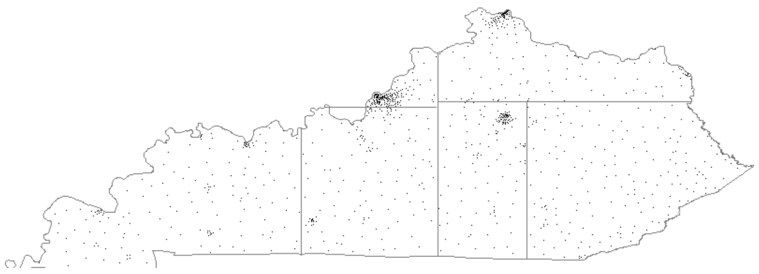

(j) Iowa

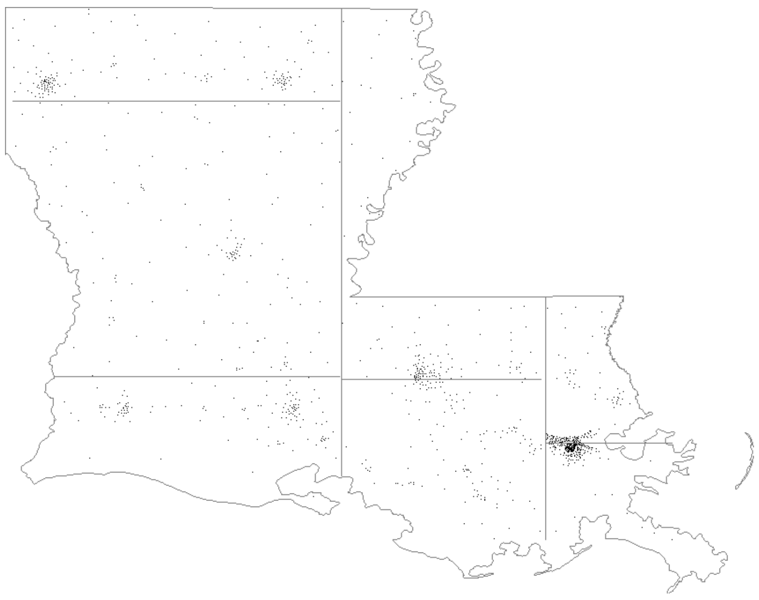

(k) Idaho

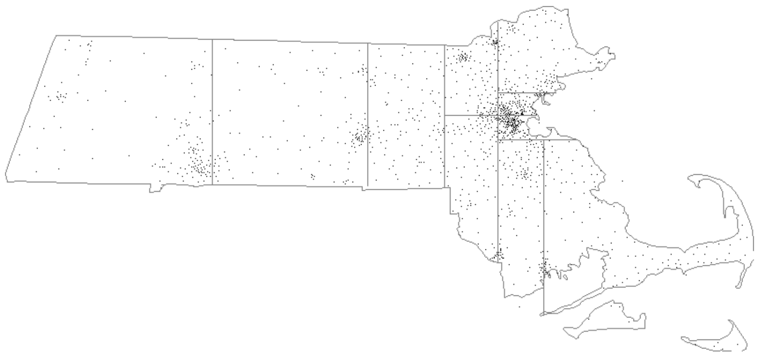

(l) Illinois


(m) Indiana

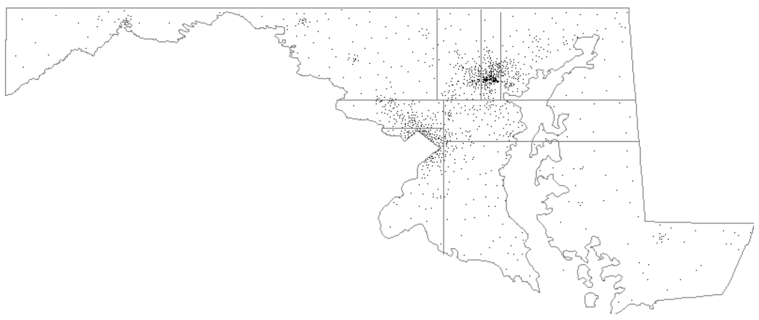

(n) Kansas


(o) Kentucky

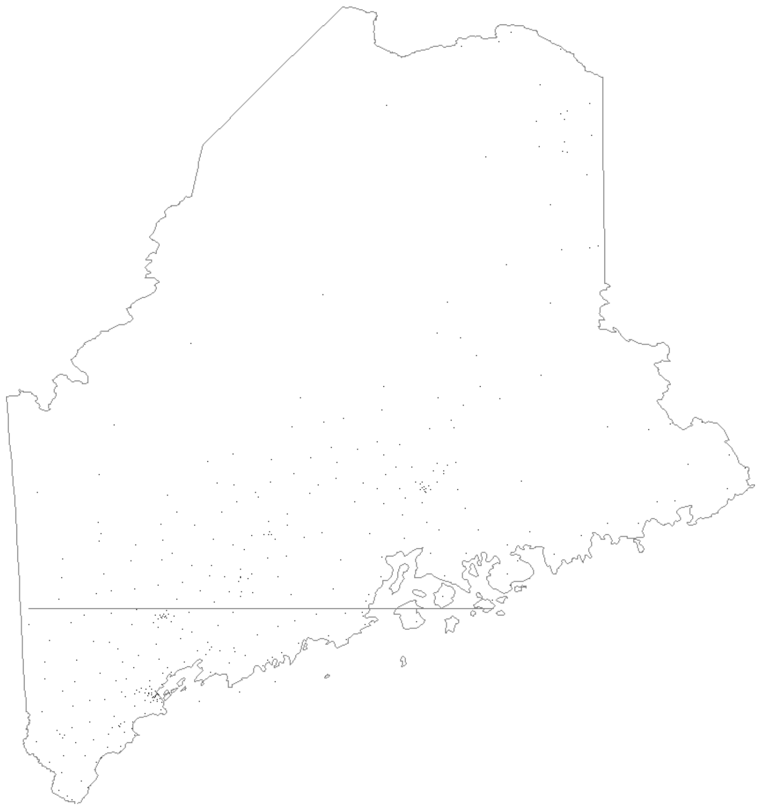

(p) Louisiana

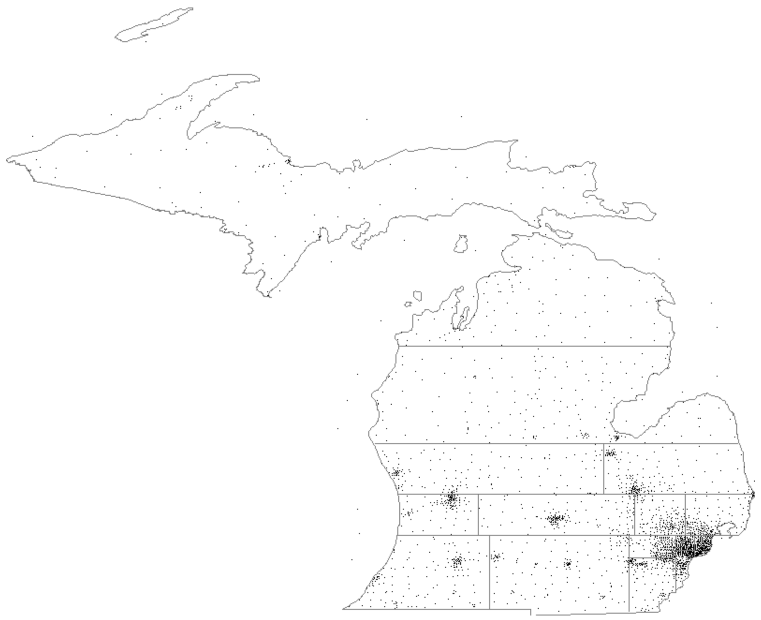


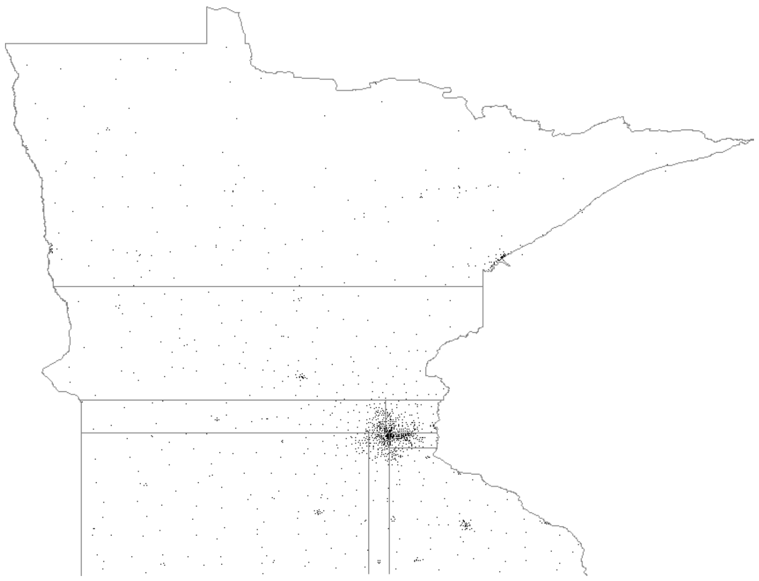
(q) Massachusetts

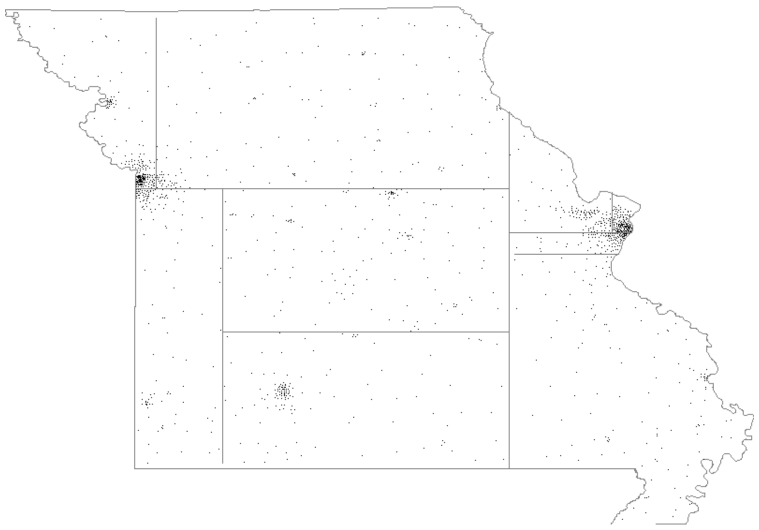


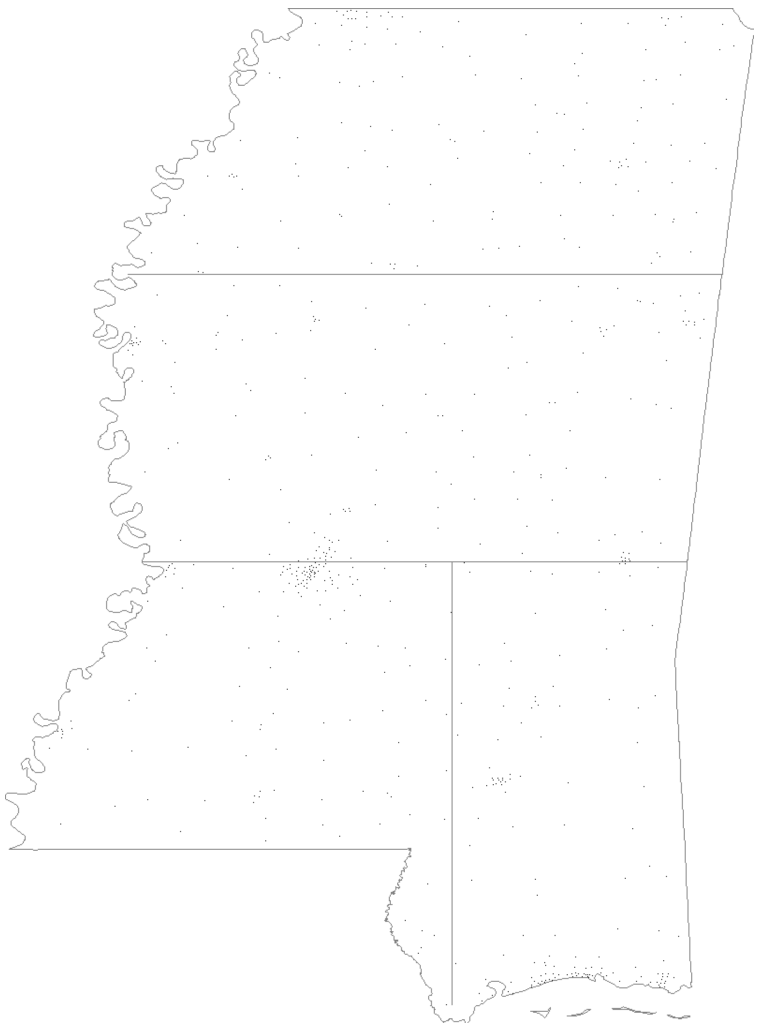
(r) Maryland

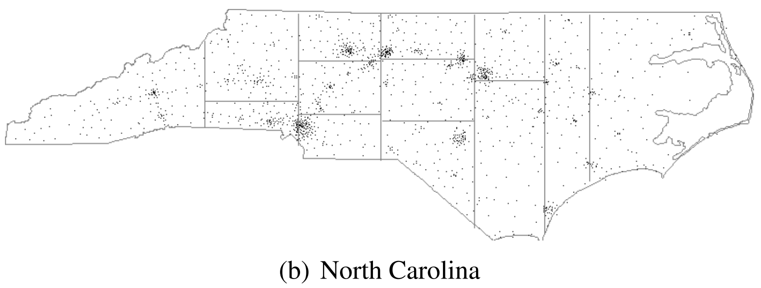

(s) Maine

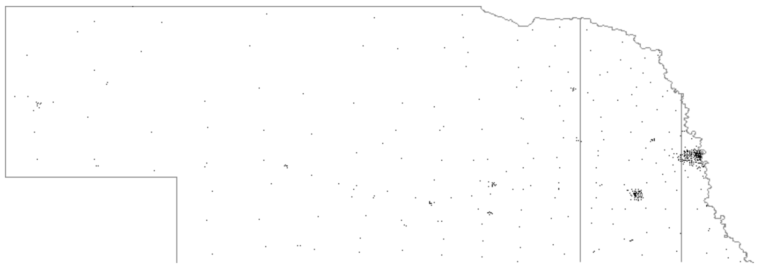

(t) Michigan

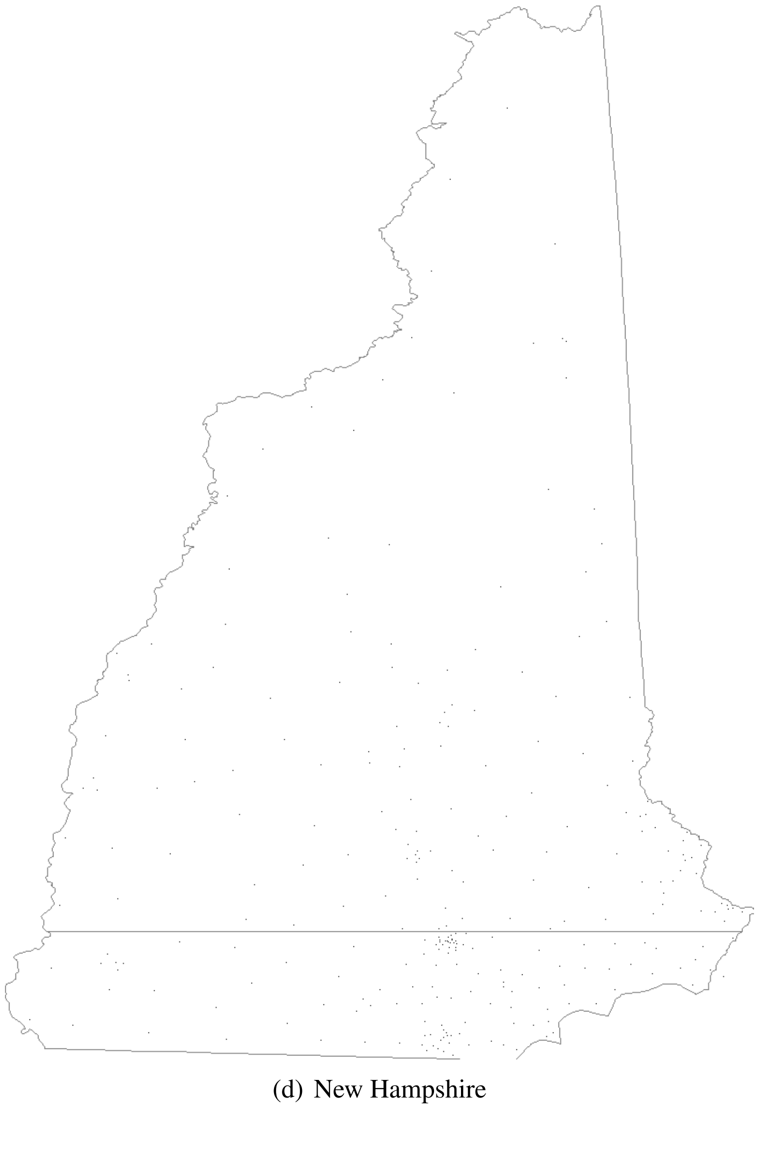

(u) Minnesota

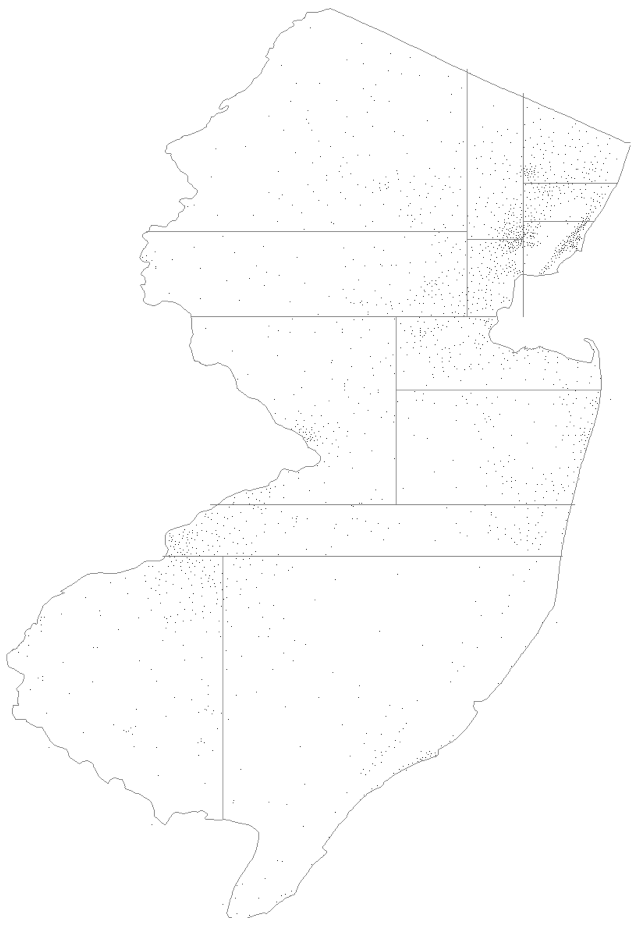

(v) Missouri

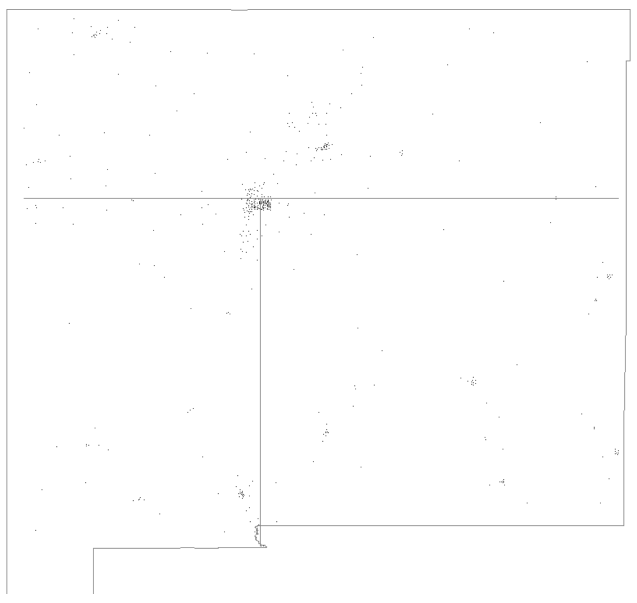

(a) Mississippi

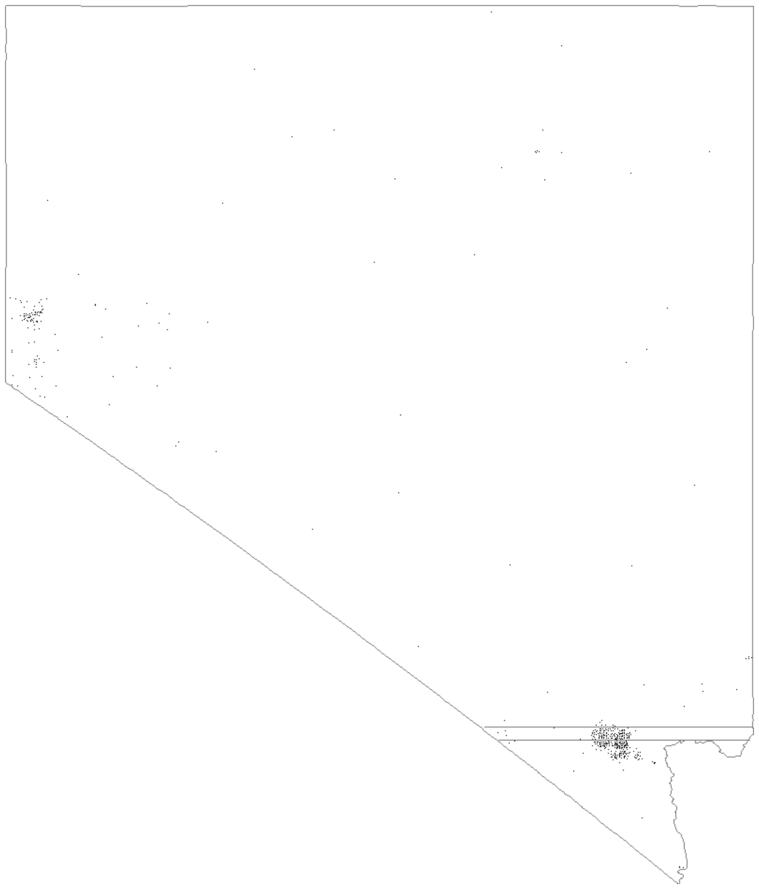

(b) North Carolina


(c) Nebraska

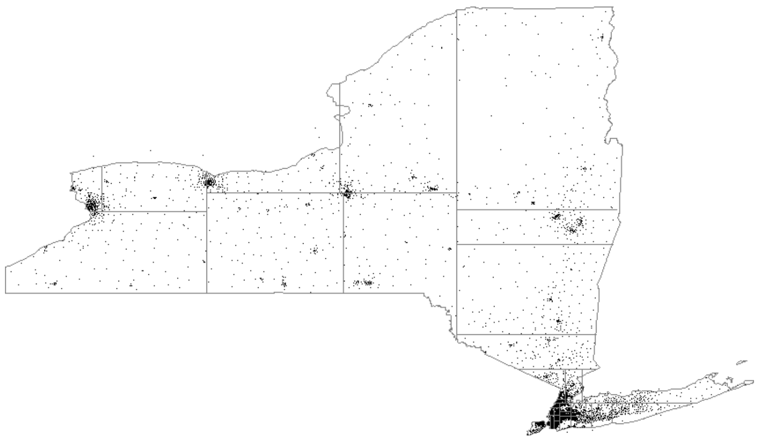

(d) New Hampshire


(e) New Jersey

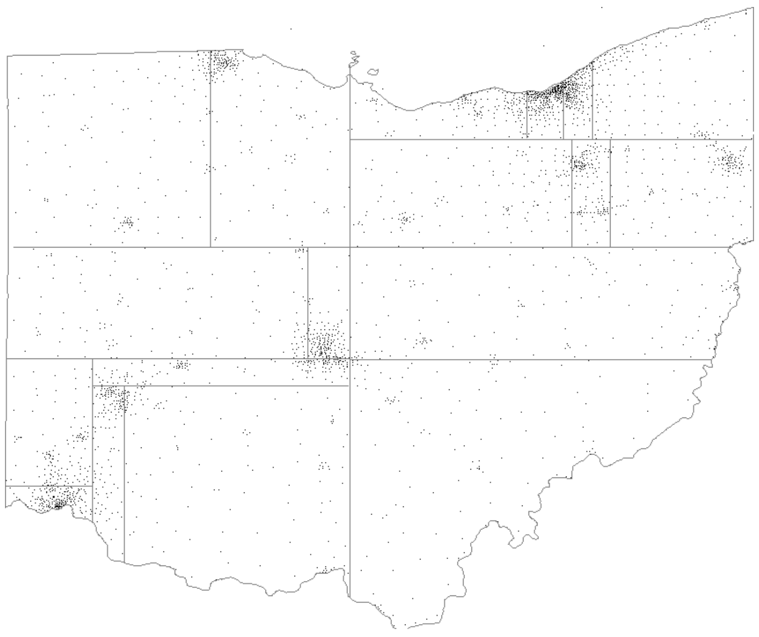

(f) New Mexico

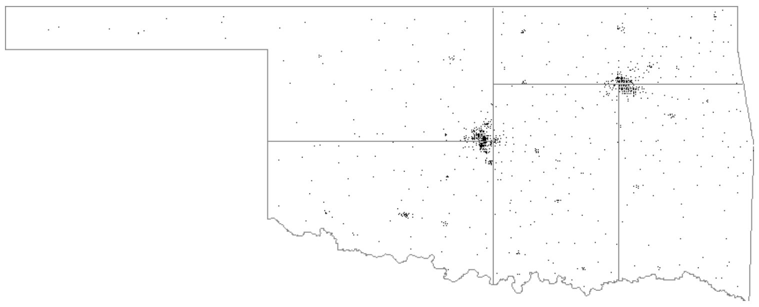


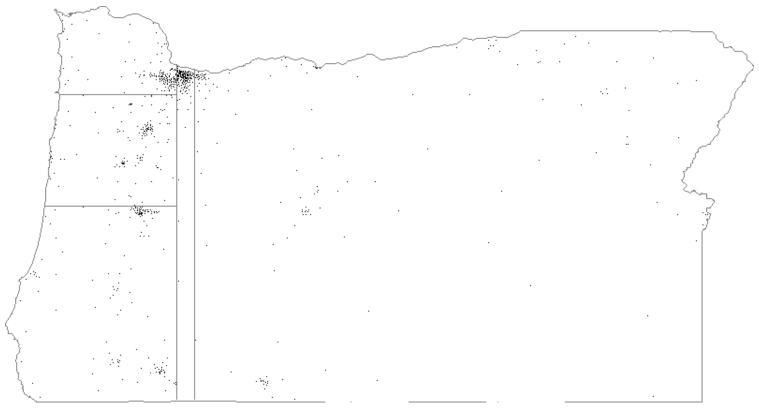
(g) Nevada

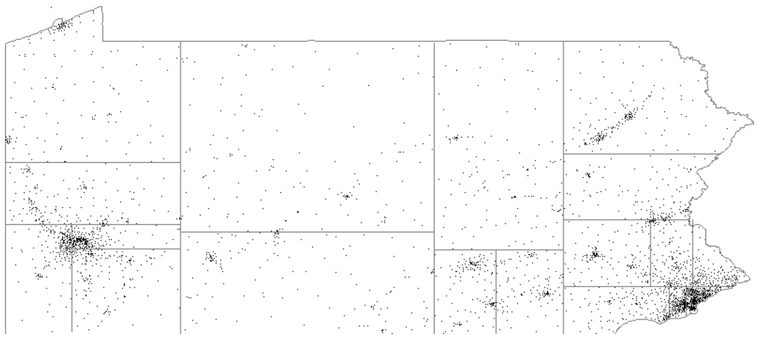


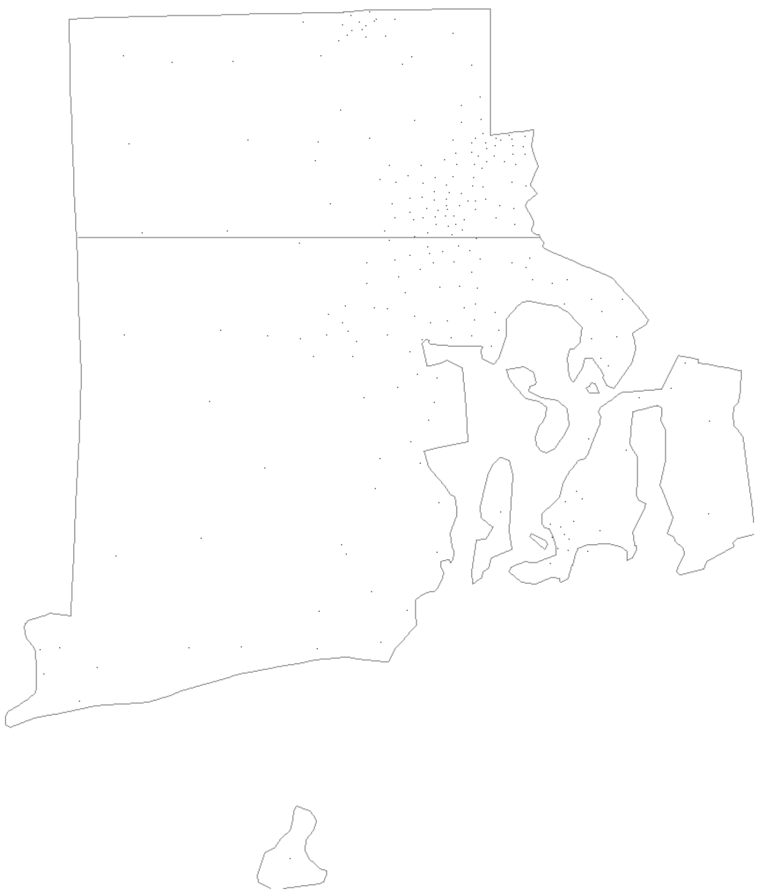
(h) New York

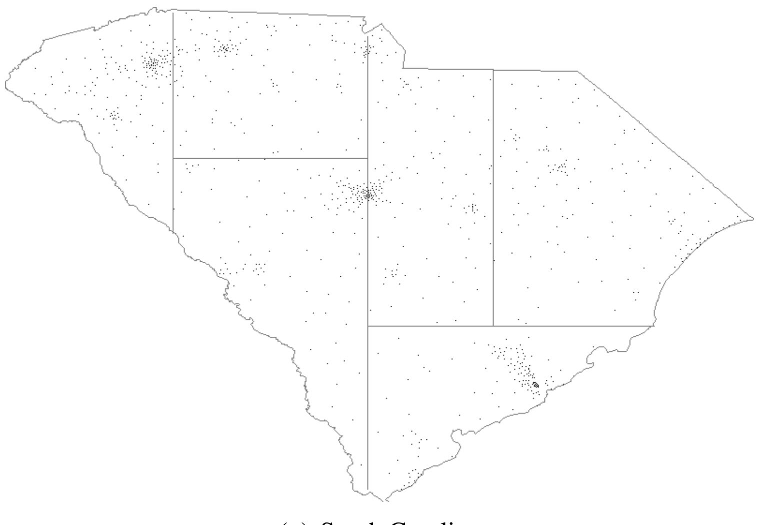

(i) Ohio

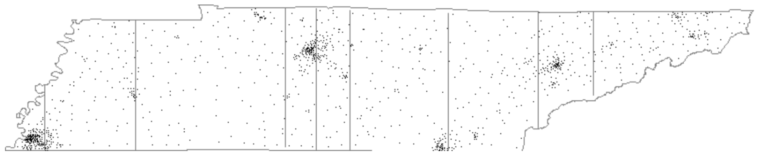

(j) Oklahoma


(k) Oregon

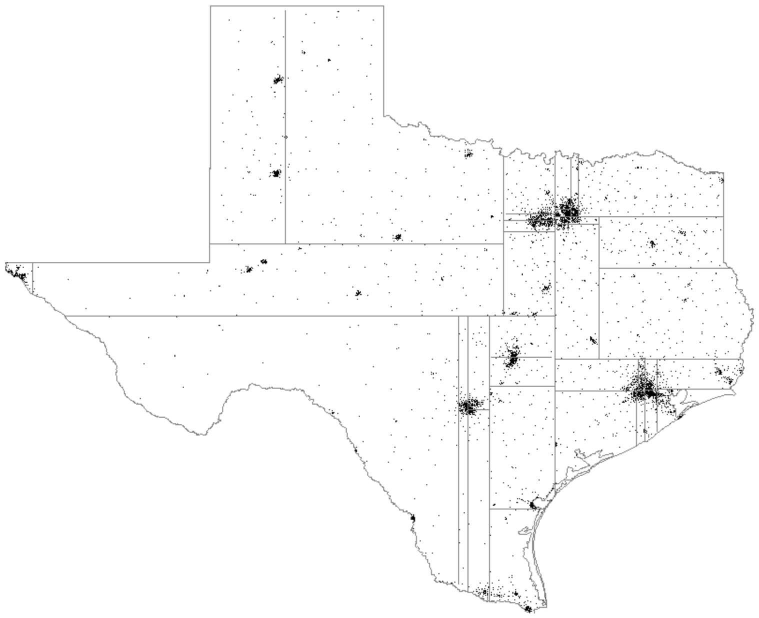

(l) Pennsylvania


(m) Rhode Island

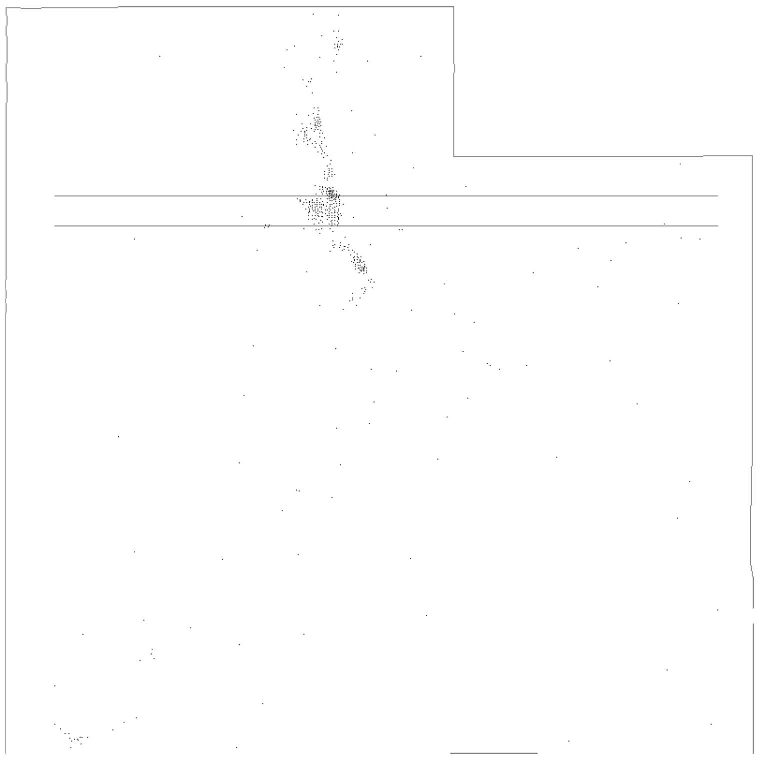

(n) South Carolina

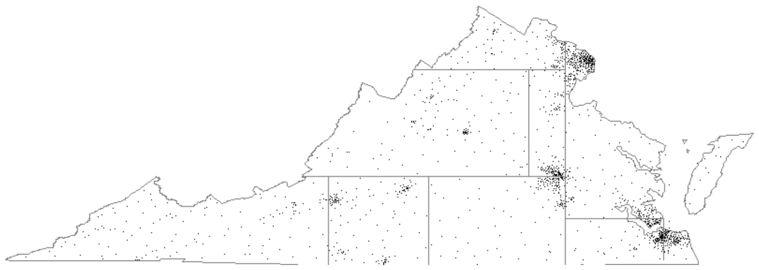


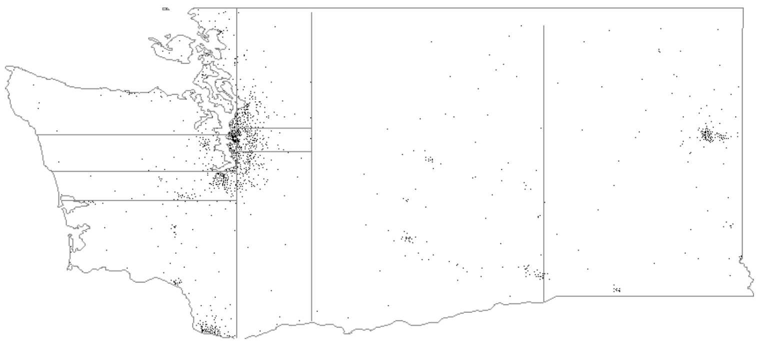
(o) Tennessee

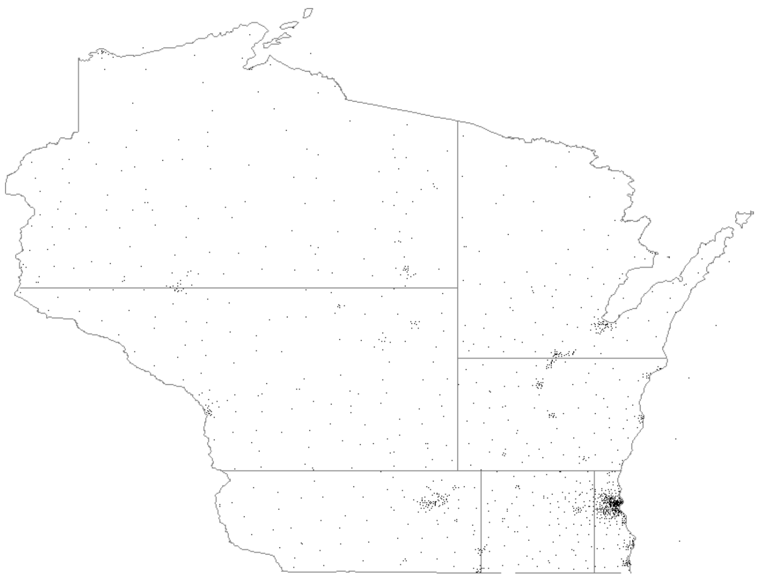


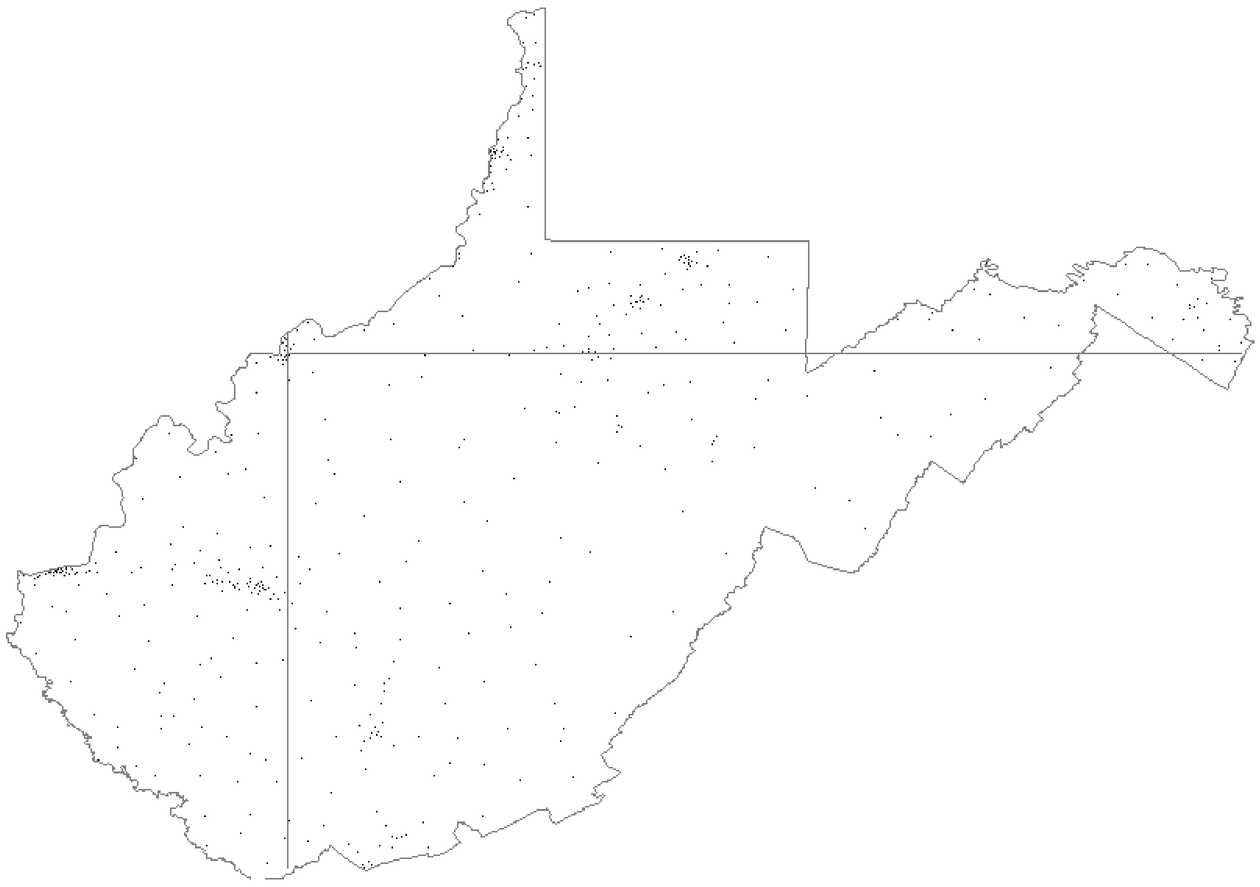
(p) Texas


(q) Utah


(r) Virginia


(s) Washington


(t) Wisconsin


(u) West Virginia