Paper 223-2010

Aggregation without aggravation: determining spatial contiguity and joining geographic areas using hashing

Gwen Babcock, New York State Department of Health, Troy, NY

ABSTRACT

Grouping smaller neighboring geographic areas into larger regions based on specified criteria was done using two SAS®9 programs. The first program creates a list of areas and their neighbors known as a contiguity table. The table is used by the second program to aggregate geographic areas into larger regions. In public health surveillance, aggregation might be necessary to preserve the confidentiality of those diagnosed with illnesses or to show stable rates in areas with sparse population. An example is provided of the merging of towns and cities in New York State until regions are created with a population of at least 20,000. SAS® allows this to be done in a systematic and objective way, using an iterative process for area aggregation. As each area is aggregated, the contiguity table is updated until all the newly created regions meet the criteria. The use of SAS hash objects allows the aggregation to be done easily and quickly, because hash objects are stored in computer memory.

INTRODUCTION

This paper describes two SAS 9.1/9.2 programs for intermediate to advanced SAS programmers. One creates a data set containing a list of geographic areas and their neighbors, called a spatial contiguity file. There are two kinds of neighbors: 'bishop' neighbors, which touch at one point, and 'rook' neighbors, which touch at two or more points. The first program identifies only 'rook' neighbors, although the code can be easily adapted to identify 'bishop' neighbors as well. The program uses the spatial contiguity file to display a map showing the number of neighbors of each area. Besides determining the number of neighbors of each area, contiguity files are useful for several other applications. They can be used for redistricting or other types of aggregation of areas. In these cases, a contiguity file limits aggregation to contiguous areas. They can be useful in statistical analysis when spatial autocorrelation might influence the variable of interest.

The second program uses the contiguity file created by the first program to aggregate smaller geographic areas using specified criteria. Such larger aggregations can be used to display health data while preserving confidentiality (El Emam et al, 2009), to create health service areas with a minimum number of patients, or to assign salses territories with a minimum number of customers or dollar value of sales. The smaller geographic areas will be called “input areas” and the newly created aggregations will be called “output regions”. This paper shows an example of using New York State minor civil divisions (towns and cities) as input areas to produce output regions. The input areas chosen are preferentially contiguous and close to one another. The output regions must have a certain minimum population and must remain within county boundaries, unless crossing the boundary is needed to meet the minimum population.

WON'T YOU BE MY NEIGHBOR: CREATING THE SPATIAL CONTIGUITY FILE

MAP DATA

SAS can be used to manipulate maps when they are available as a special data set called a map data set. Map data sets contains the information needed to draw map boundaries: x and y coordinates along with a variable that identifies the area. The use of these data sets to create maps has been described elsewhere (Zdeb, 2004). For this example, we will use a map of New York State cities and towns (CSCIC, 2007). SAS provides many map data sets, and others are available in the format of a ESRI Shapefile file set, which is a popular geospatial vector data format for Geographic Information Systems software. PROC MAPIMPORT is used to import a shapefile, and turn it into a SAS map data set. The strategy will be to sequentially compare all of the x,y coordinates that define each area in the map to all of the other x,y coordinates in the map. If a pair of coordinates are the same but the areas they define are different, the areas must be bishop's neighbors. If an additional shared pair is found between the two areas, the areas must be rook's neighbors. In order to do this comparison rapidly, a copy of the map will be placed in a hash object in memory, and this copy will be compared to the original data set.

The “town” map data set contains four variables characteristic of all map data sets: “ID”, “SEGMENT”, “X”, and “Y”, in addition to other variables unique to it which will be used later. The variable “ID” contains the Federal Information Processing (FIPS) codes which uniquely identify each area. The “X” and “Y” variables identify the coordinates of each node. The “SEGMENT” variable is not used in this analysis.

The first part of the program takes care of some housekeeping. We tell SAS where the map data set is and where we want our results to be stored using LIBNAME statements. The variable in the map data set that uniquely identifies each area is assigned to the "IDVAR" macro variable. Lastly, will will use PROC MAPIMPORT to create the SAS map data set from the shapefile.

```
libname cdisk "C:\My Documents\My SAS Files\9.1\SGF\2009\";
%let confile=mcdcont;
%let idvar=id;

proc mapimport
   datafile="C:\My Documents\My SAS Files\9.1\SGF\2009\town_small_region.shp"
   out=mymap;
run;
```

PREPARING THE MAP DATA SET AND ITS HASH OBJECT COPY

Having duplicate nodes for the same area will cause problems in later steps, so they are removed using PROC SORT with the NODUPKEY option. For the hash object, we want to retrieve a node by it's x,y value. However, the x,y values are not a unique key, which is required for hash objects in SAS 9.1. Therefore, we will add a variable 'z' such that each x,y,z is unique for each area. We will also remove missing values, which SAS uses to delineate the beginning of 'holes' in areas, but are not needed here. A 'hole' might occur when mapping a town that is completely surrounded by another town or city, or when mapping land and the land area completely surrounds a water body, like a lake or pond. The maximum value of variable 'z' is then determined using PROC MEANS and put into the macro variable 'maxnb'.

```
proc sort data=mymap out=mymap2 nodupkey; by x y &idvar.;run;

data hashmap (keep=x y z &idvar.);
retain z;
set mymap2(where=(x NE . and y NE .));
by x y;
if first.x or first.y then z=1;
z=z+1;
run;

/*determine the maximum value of z*/
proc means data=hashmap noprint;
output out=maxz max(z)=mz; run;
/*put the maximum value of z into macro variable maxnb*/
data _null_;
set maxz;
call symputx("maxnb",mz);
run;
```

THE HEAVY LIFTING, MADE EASY USING HASHING

Now that the data are set up, we can determine who is a neighbor of who. To do this, we set up three hash objects. One contains the entire map data set, and allows us to look up the data by x,y value. The second hash object will hold a list of rook neighbors (which share at least two points with the area of interest) for all areas. This second hash object holds the results we want to output. These hash objects need be instantiated only once. The final hash object will hold a list of bishop neighbors (neighbors which share at least one point) for the area of interest. This last hash object is instantiated for each area.

Once the hash objects are set up, we use a SET statement to within a DOW loop (Dorfman,2008; Dorfman and Shajenko, 2007; Dorfman and Vyverman, 2009) to read in the map again, by area of interest. We need to rename the variables so that there is no confusion with the variables of the potential neighbor which will be pulled from the hash object. A flag is set up to keep track of whether we've found any neighbors or not; we want all areas to be represented in the final data set, even those with no neighbors. Areas with no neighbors will not be added to the hash object, so we output them into a separate data set called 'nonb'.

We cycle through each area point by point. For each point, we use the FIND method to look up in the map hash object to see if there are any areas with the same x,y coordinates. Such an area must be identical to the area in question or be a bishops or rook's neighbor of the area in question. If the area is not identical to the area in ques-

tion, a further check is done to see if it is already in the 'bishops' hash object. If it is not already in the 'bishops' hash object, it is added to the 'bishop's' hash object. If it is already in the 'bishops' hash object, this means that the area has two points in common with the area of interest, and is a rook's neighbor. In this case, it is added to the 'rooks' hash object and the flag is set. Once all the points in each area are examined, if the flag has not been set, the observation is output to the 'nonb' data set. This might happen, for example, if the area in question is an island. Once all the areas are examined, the 'rooks' hash object is turned into a hash data set using the OUTPUT method.

```
/*for each point in the map, determine the id's of neighbors which
also have the same point*/
proc sort data=mymap2; by &idvar.;run;

data nonb (keep=myid);
length x y z 8; format &idvar. $16.; /*use a format statement for character id
values*/
if _n_=1 then do;
   /*this hash object holds a second copy of the entire map for comparison*/
   declare hash fc(dataset: "hashmap");
   fc.definekey("x","y","z");
   fc.definedata("x","y","z","&idvar.");
   fc.definedone();
   call missing (x,y,z,&idvar.);
   /*this hash object will hold the rook neighbors for each area: they have two
points in common*/
   declare hash rook();
   rook.definekey("&idvar.","myid");
   rook.definedata("&idvar.","myid");
   rook.definedone();
end;
   /*this hash object holds the bishop neighbors for each area: they have a point
in common*/
   declare hash bishop();
   bishop.definekey("&idvar.","myid");
   bishop.definedata("&idvar.","myid");
   bishop.definedone();
foundnb="N";
do until (last.myid);
   set mymap2 (keep=&idvar. x y rename=(&idvar.=myid x=myx y=myy) where=(myx NE .
and myy NE .)) end=eof;
   by myid;
   do n=1 to &maxnb.; /*this is max number of points in common =max z*/
         rc=fc.find(key:myx, key:myy, key:n);
         if rc=0 and myid NE &idvar. then do;
               nbrc=bishop.check(key:&idvar, key:myid);
               if nbrc=0 then do;
                     rc2=rook.add(key:&idvar, key:myid, data:&idvar, data:myid);
                     foundnb="Y";
               end;
               else rc1=bishop.add(key:&idvar, key:myid, data:&idvar, data:myid);
         end;
   end;*do &maxnb.;
end;*end DOW loop;
if foundnb="N" then output nonb;
if eof then rook.output(dataset:"cdisk.&confile.");
run;
```

THE RESULTS OF THE FIRST PROGRAM

The result is a data set ("mcdcont") containing one observation for each unique area-neighbor pair. Table 1 shows a partial printout of the formatted data set. A secondary result is the output data set 'nonb' which contains a

list of those areas with no neighbors. To use these data sets to create a map of the number of neighbors, there are three steps. Firstly, PROC MEANS is used to determine the number of neighbors of each area. Secondly, areas with no neighbors are added back in using a DATA step. Lastly, PROC GMAP is used to create the map.

Table 1: Output of the spatial contiguity program, consisting of a list of areas and their 'rook' neighbors.

myid	id
001\01000	083\61148
001\01000	083\52100
001\01000	001\17343
001\01000	001\31104
001\01000	001\06354
001\01000	083\22117
001\06211	095\10154
001\06211	095\46855
001\06211	001\79851
001\06211	001\40002
001\06211	001\50672
001\06211	095\83195
001\06211	001\61181
001\06354	001\16694
001\06354	083\65541
001\06354	001\31104
001\06354	083\22117
001\06354	001\01000
001\06354	001\50672
001\16694	039\49935

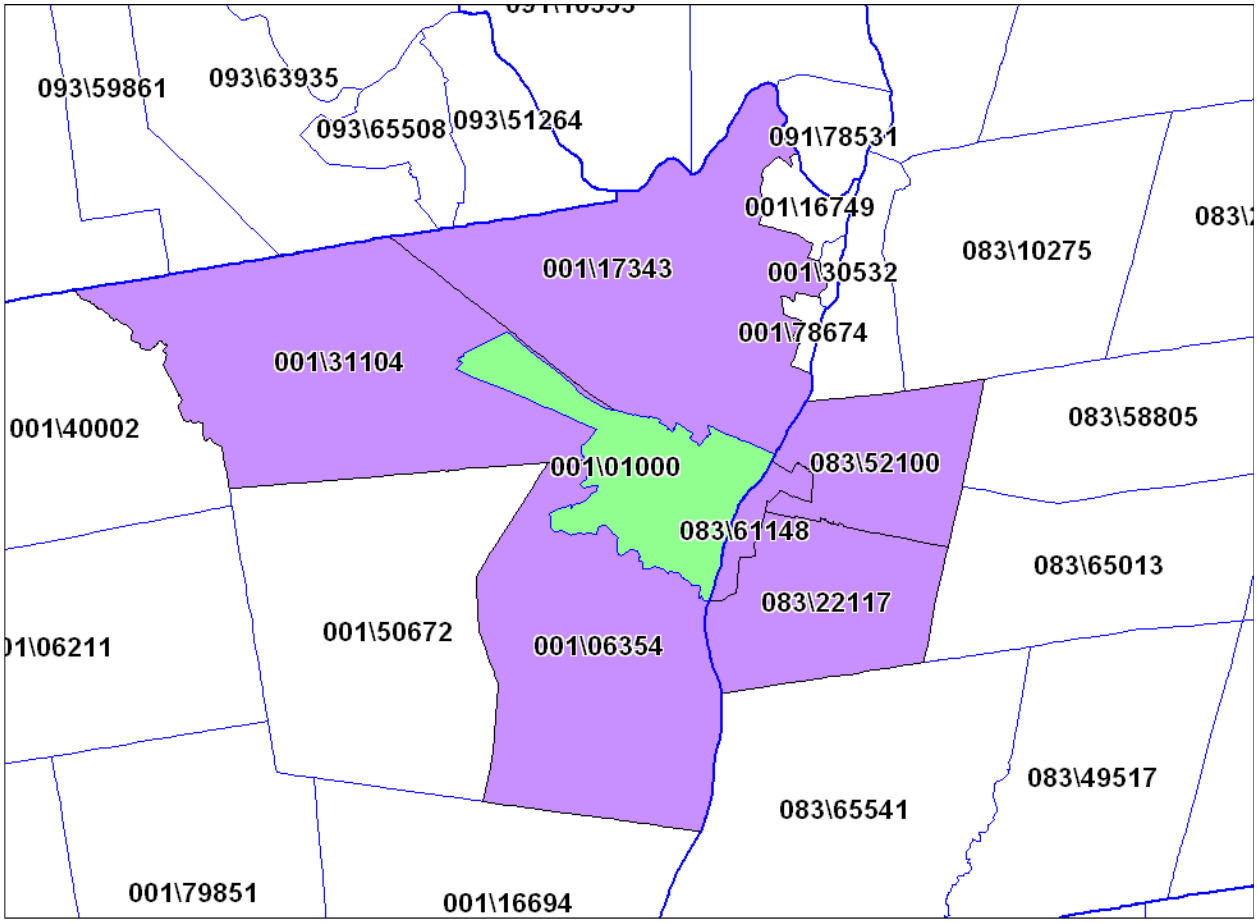

Figure 1: Map showing area 001\01000 (the City of Albany) and its 'rook' neighbors.

```
/*create a data set with number of neighbors*/
proc freq data=cdisk.&confile. noprint;
table myid /out=numnb (drop=percent rename=(count=countnb)) missing;
run;

data numnball ;
set numnb (rename=(myid=&idvar)) nonb (rename=(myid=&idvar));
if countnb=. then countnb=0;
run;

proc gmap map=mymap data=numnball;
choro countnb / discrete;
id &idvar.;
label countnb="Number of neighbors";
run;quit;
```

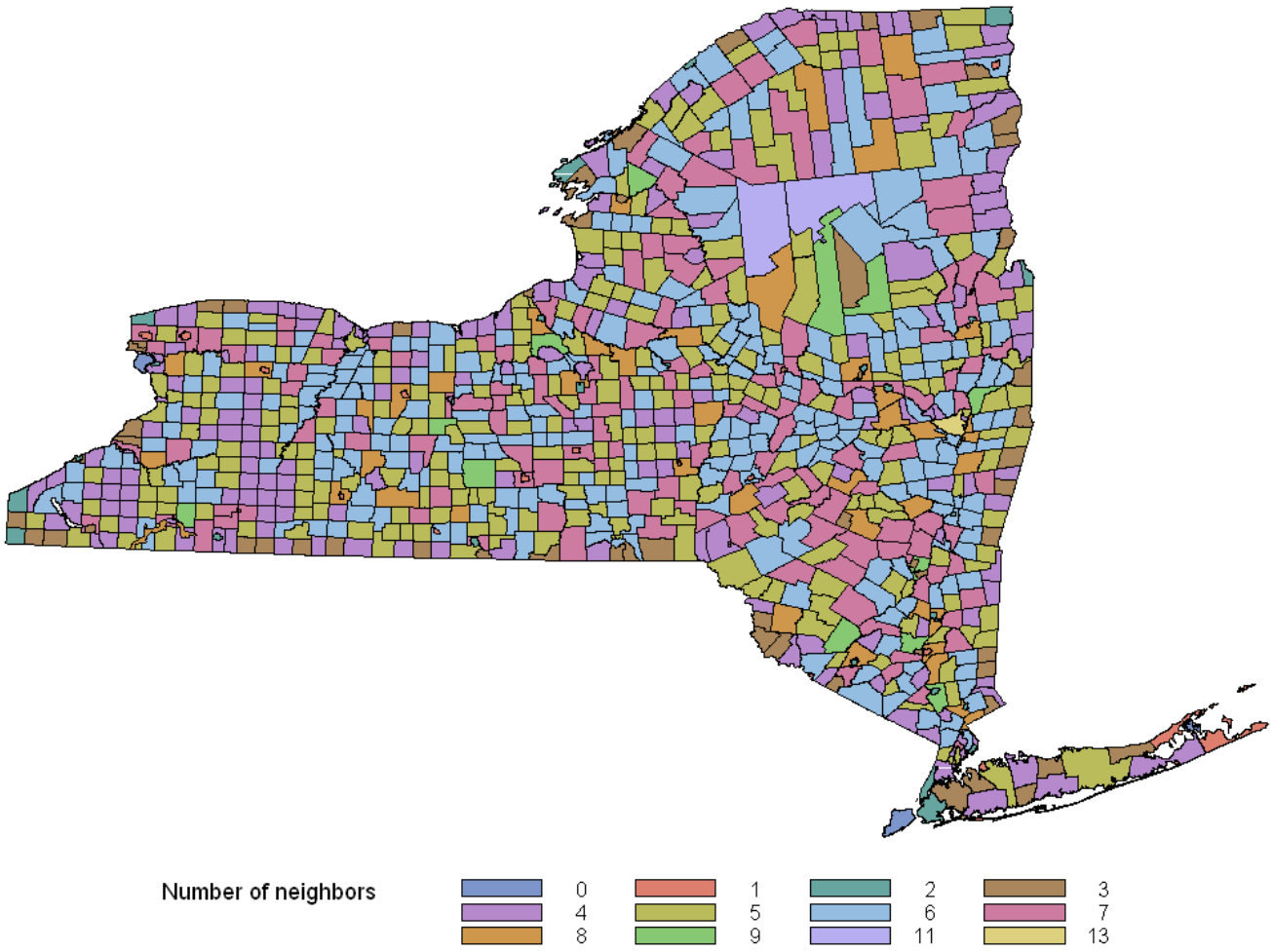

Figure 2: Output from the SAS 9.2 GMAP procedure: chloropleth map of New York State Towns showing number of neighbors. This map was created with information copyrighted by the New York State office of Cyber Security and Critical Infrastructure Coordination © 2007.

AGGREGATING GEOGRAPHIC AREAS

Once the contiguity table has been created, it can then be used in the second program to aggregate geographic areas. In this example, New York towns and cities will be aggregated such that each new region has a minimum population of 20,000. Some towns and cities are already be above this limit, and therefore might not be aggregated.

TOWN/CITY DATA

For the aggregation, besides the variables described earlier, this data set must contains at least four other variables (Table 2). It must contain a variable containing the population of the town/city, a variable containing the county of the town/city, and variables containing the latitude and longitude of the town/city in decimal degrees. In this example, the latitude and longitude represent geographic centroids. Zdeb (2002) describes a method for calculating a latitude/longitude of the visual center of a map object, or the %CENTROID macro can be used for this purpose (SAS Institute Inc, 2008).

Table 2: Minimum variables needed in the towns data set for aggregation.

NAME_SHORT	ID	COUNTY	TOTAL_POP	LATITUDE	LONGITUDE
Albany	001\01000	001	95658	42.6598	-73.7813
Berne	001\06211	001	2846	42.5974	-74.1236
Bethlehem	001\06354	001	31304	42.6021	-73.8267
Coeymans	001\16694	001	8151	42.4935	-73.8411
Cohoes	001\16749	001	15521	42.7733	-73.7031
Colonie	001\17343	001	79258	42.7286	-73.7845
Green Island	001\30532	001	2278	42.7442	-73.6942
Guilderland	001\31104	001	32688	42.7013	-73.9252
Knox	001\40002	001	2647	42.6891	-74.1029
New Scotland	001\50672	001	8626	42.6210	-73.9350

CONTIGUITY DATA

This data set contains a list of areas and their neighbors (Table 1). Neighbors are those areas which are candidates for merging, in this example chosen because they are geographically contiguous (Figure 1). It is also possible to use a contiguity file based on factors other than geographic adjacency, such as language, etc.

PREPARING THE CONTIGUITY DATA SET AND THE DATA FILE

First, we use a LIBNAME statement to tell SAS where to find the data. Macro variables are used to hold important information. The macro variable 'datafile' is the name of the data set containing an id, population, county, and the latitude and longitude of the centroid for each town or city in New York StateThe macro variable 'idvar' is the variable which uniquely identifies each input area. The 'criteriavar' variable holds a number which will be summed over the input areas to determine when sufficient aggregation has occurred to create an output region. In this example, the 'criteriavar' is the variable total_pop, which is the 2000 Census population counts for each town. The 'mincriteria' variable is the minimum value of the 'criteriavar' needed for each output region. In this example, I have decided that each output region should have a minimum population of 20,000.

Next, the town information data set and the contiguity file are sorted by their unique identification variables, so they can be used with 'BY' statements later. Then, a DATA step is used to add an arbitrary sequential number ('keyno') for each input area's neighbors to the contiguity file (Table 3). This prepares the file for use as a hash object later. The original contiguity file contains multiple records for each input area, but a hash object (in SAS 9.1) requires a unique key. The sequential number, in combination with the input area id, provides a unique key.

```
libname cdisk "C:\My Documents\My SAS Files\9.1\SGF\2009"
%let datafile=towns;
%let idvar=id;
%let criteriavar=total_pop;
%let mincriteria=20000;

proc sort data=cdisk.&confile; by myid;run;
proc sort data=cdisk.&datafile; by &idvar.; run;

/*add arbitrary sequential number to use as hash key later*/
data conarealist (rename=(&idvar=neighbor myid=id));
set cdisk.&confile.;
by myid;
retain keyno;
keyno=sum(keyno,1);
if first.myid then keyno=1;
run;
```

Table 3: Contiguity file with sequentially numbered neighbors.

id1	id2	keyno
001\01000	001\06354	1
001\01000	001\17343	2
001\01000	001\31104	3
001\01000	083\22117	4
001\01000	083\52100	5
001\01000	083\61148	6
001\06211	001\40002	1
001\06211	001\50672	2
001\06211	001\61181	3
001\06211	001\79851	4

Following that, the number of neighbors is added to the town information data set. This is done so that areas with fewer neighbors can be merged before areas with more neighbors. The number of neighbors was previously determined by running the FREQ procedure on the contiguity file. This is merged back into the town information data set using another DATA step.

```
/*put number of neighbors in main dataset*/
proc sort data=numnball; by id;run;

data allareas (rename=(latitude=lat longitude=lon));
merge cdisk.&datafile.(rename=(&idvar.=id)) numnball;
by id;
run;
```

AGGREGATION USING HASH OBJECTS

Now that the data are set up, the geographical aggregation can be performed. The entire aggregation takes place in the first iteration of the DATA step. (Subsequent iterations will assign a final output region to each input area.) The input areas will be aggregated pairwise until the output region satisfies the criteria. After each pairwise aggregation, the newly formed output region becomes a candidate input area for the next pairwise aggregation. There are five hash objects used, three for various inputs to the aggregation DATA step and two for outputs (Table 4). The largest hash object ('blist') is the town/city data; its key is the unique id of the town/village. This hash object functions like a phone directory, allowing us to obtain information about the input areas whenever we need it. A second hash object ('ord') also holds the same town/city data, but it's purpose is different; this hash object is sorted, and allows us to select the highest priority for area for merging. Accordingly, the key is different than for 'blist': the key is the total population, number of neighbors, and unique id. We create a corresponding hash iterator object so we can retrieve these observations sorted by the key: the lowest population first, the highest last. If there are ties in population size, the area with the fewest number of neighbors would be retrieved. If there is also a tie in numbers of neighbors, the area with the lowest id would be retrieved; in this case, that is an arbitrary order. The last input hash object ('nb') contains the data from the contiguity file. This hash object allows us to find candidates for merging with the highest priority area. The first and main output hash object ('idregion') contains the records of the pairwise merges, which consists of the unique identifiers of the two input areas and the output region. The second output hash object ('nl') holds a list of the neighbors of the current output region, which is used to update the contiguity hash object and to determine the total number of neighbors, which is used to update the input hash objects which contain that information. The second hash output object is recreated for each pairwise aggregation.

After the hash objects are set up, a large DO loop is set up. This loop controls the extent of the aggregation: pairwise aggregation will continue UNTIL the criteria is met, in this case, when the population is greater than 20,000. Within the DO loop, the strategy for pairwise aggregation is as follows: 1) the input area with the smallest population is selected for aggregation 2) the input area's neighbors are examined and the desired neighbor is selected to be the second input area 3) the two input areas and the new output region's unique identifiers are stored in hash object 'idregion' 4)a list of neighbors of the new region is created in hash object 'nl' 5) the output region's data are

added to hash objects 'blist' and 'ord' and data about the input regions are removed from 'ord' and 'blist' 6)the contiguity table is updated to reflect the new map boundaries 6)pairwise aggregation is repeated until the total population is at least 20,000.

Table 4: Hash objects used in the aggregation program

Hash object	Key(s)	Data	description
blist	Unique id of city/town	Population, county, number of neighbors, centroid coordinates	Allows us to get information about a town/city if we know it's id
ord	Total population, number of neighbors, unique id of city/town	Total population, number of neighbors, unique id of city/town	This hash table is sorted, and provides the first member of the pair to be merged
nb	Unique id of city/town, sequential number of neighbor	Unique id's of neighboring city(ies)/town(s)	Contains the spatial contiguity file: a list of cities/towns and their neighbors
nl	Unique id of city/town	Unique id of city/town	A list of neighbors of the newly created output region
idregion	Unique id of input area	Unique id of output region	This hash table holds the results of the pairwise merges, two records per merge

In the first part of the DATA step, the KEEP data set option ensures that the final data set is not cluttered with extraneous variables. Next, a FORMAT statement assigns formats to variables, ensuring that SAS will not assign incorrect default lengths to derived variables and that it knows which variables are character and which numeric. The IF-THEN statement ensures that the statements preceding the do loop are only performed when _n_=1, i.e., during the first iteration of the DATA step. During this iteration, the hash objects are set up using DECLARE statements followed by the DEFINEKEY, DEFINEDATA, and DEFINEDONE methods. Three hash iterators are also set up using DECLARE statements. These allow us to access the hash objects in sequential order.

```
data finalregions (keep=newregion oldid county &criteriavar. countnb);
format id oldid nbid neighbor charreg newregion $10. county newcounty nbcounty
$15. lat lon 16.8;
if _n_=1 then do;
   /*set up hash objects*/
   declare hash blist(dataset:'allareas'); /*this is a list of all areas with
info by id*/
   blist.definekey('id');
   blist.definedata('id', "&criteriavar.", 'county', 'lat', 'lon', 'countnb');
   blist.definedone();
   declare hash idregion();
   idregion.definekey('id');
   idregion.definedata('id','newregion');
   idregion.definedone();
   declare hash ord(dataset:'allareas', ordered: 'a');
   ord.definekey("&criteriavar." ,'countnb','id');
   ord.definedata("&criteriavar." ,'countnb','id');
   ord.definedone();
   declare hash nb(dataset:'conarealist'); /*this is the contiguity hash object*/
   nb.definekey('id', 'keyno'); /*must have unique key or values will be replaced
unless use multidata option in SAS 9.2*/
   nb.definedata('id','keyno' ,'neighbor');
   nb.definedone();
   declare hiter rord('ord');
   declare hiter ridregion('idregion');
   declare hiter rblist('blist');
   call missing(&criteriavar., lat, lon, countnb, id, neighbor, newregion,
keyno); /*only need for missing vars*/
```

```
firstcases=0; region=0;    newcases=0;
```

The second part of the DATA step is where the pairwise merges occur. These merges need to continue until the population is at least 20,000, which is ensured by nesting them in a DO UNTIL loop. Within this loop, the first input area to be aggregated is chosen using the FIRST method on the 'rord' iterator. This selects the area with the lowest population for merging. For later steps, will need to know certain information about this input area. This information is obtained by using the FIND method on the 'blist' object using as the key the identification variable obtained from the FIRST method. This information is then stored in variables; I used the prefix 'first' for these variable names to help me distinguish this information from the information about other input areas. The new region to be formed is given a unique id, stored in numeric form in the variable 'region' and as a character in the variable 'charreg'.

```
do until (firstcases GE &mincriteria.);
 rc=rord.first(); /*choose the input area to be aggregated*/
 region=region+1; charreg=left(put(region,8.));
 rc=blist.find(key:id); /*get info about block/region to be merged*/
 /*first, keep certain values from our block of interest*/
 firstid=id; firstcases=&criteriavar.;  firstcountnb=countnb;
 firstlat=lat; firstlon=lon; firstcounty=county;
 if firstcases=. then putlog _all_; x=0;
```

```
%macro geodist (lat1,long1,lat2,long2);
  /**6371 is the earth's radius in kilometers**/
  %let pi180 = 0.0174532925199433;
  2*6371*arsin(sqrt((sin((&pi180*&lat2-
&pi180*&lat1)/2))**2+

cos(&pi180*&lat1)*cos(&pi180*&lat2)*(sin((&pi180
*&long2-&pi180*&long1)/2))**2));
%mend;
```

Figure 3: Geodist macro, calculates distance between two

Next, the desired neighbor to be merged must be found. This neighbor will be the second input area which will be merged with the first input area to form an output region. If the first input area has contiguous neighbors, they will be candidates to become the second input area; they are examined sequentially by using the FIND method on the 'nb' hash object, which contains the contiguity table. If the input area has no contiguous neighbors, then all of the other areas become candidates; they are examined sequentially by using the FIRST and NEXT methods on the 'rblist' hash iterator. Once the candidate is chosen, the distance between the candidate and the first input area is calculated using the %geodist macro (Figure 3, Ward) (in SAS 9.2, the GEODIST() function could be used instead). If the candidate second input area is the first one considered, it is tentatively chosen as the second input area. Additional candidates are compared to the previously chosen candidate by a series of IF-THEN statements. Most of the time, the closest candidate in the same county will be chosen as the second input area. If there is no candidate second input area in the same county, the closest area will be chosen. Information about the second input area is stored in variables with the prefix 'nb'.

```
*************************************************************************
/*for the input area to be aggregated, find a neighbor that meets our criteria*
*************************************************************************;
do until(rcn NE 0);
   x=x+1;
   if firstcountnb>0 then do;
                rcn=nb.find(key:firstid,key: x); /*this gives us the id of the
neighbor x*/
                rc=blist.find(key:neighbor); /*get info about neighbor x*/
   end;
   else do;
         /*if there is no contigous neighbor, look through all the areas*/
         if x>1 then rcn=rblist.next();
         else if x=1 then do; rcn=rblist.first();
               putlog charreg= firstid " does not have contiguous neighbors";
         end;
         if id=firstid then rcn=rblist.next();/*cannot be own neighbor*/
         neighbor=id;
   end;
   if rcn=0 then do; /*if found a neighbor*/
```

```
   /*calculate distance.  in 9.2, you can remove the % and use the geodist func-
tion*/
         distance=%geodist(lat,lon,firstlat,firstlon);
/*This section compares the current neighbor to the 'best' one seen so far
         to see if the current neighbor is better*/
   /*if this is the first neighbor, take it unless we find a better one*/
   if x=1 then do;
         nbid=neighbor; nbcounty=county; nbdist=distance;
   end;
   /*a neighbor would be better if it were in the same county*/
   if firstcounty NE nbcounty and firstcounty=county then do;
         nbid=neighbor; nbcounty=county; nbdist=distance;
   end;
   /*or to be closer and in the same county*/
   if firstcounty=nbcounty and firstcounty=county and distance<nbdist then do;
         nbid=neighbor; nbcounty=county;  nbdist=distance;
   end;
   /*or just to be closer*/
   if firstcounty NE nbcounty and firstcounty NE county and distance<nbdist then
   do;
         nbid=neighbor; nbcounty=county; nbdist=distance;
   end;
   end; /*end if found neighbors*/
end;
*************************************************************************
/*this ends the section for finding the neighbor that meets the criteria*
*************************************************************************;
```

Once the two input areas have been chosen, the output region can be created. Information about the output region will be placed in variables with the prefix 'new'. First, we obtain the number of neighbors and population of the second input area using the FIND method on the 'blist' object. Next, we calculate the number of cases and the latitude/longitude of the output region. Lastly, we determine the number of neighbors of the output region. We assume that neighbors of the input areas (excluding the input areas themselves) will be neighbors of the output region. We get the neighbors by using the FIND method on the contiguity hash object 'nb', using the input areas' ids as part of the key. Then, we list all the neighbors by creating a hash object 'nl' and populating it with the neighbors of both input regions using the ADD method. Lastly, we find the number of neighbors using the NUM_ITEMS method on the hash object 'nl'. Notice that even if two input areas share a neighbor, there will be only one entry in the 'nl' hash object for this neighbor, because the neighbor's id is used as the key, and the default behavior of hash objects is to contain only one entry per unique key. Thus, the list of neighbors is automatically deduplicated.

```
rc=blist.find(key:nbid); /*get the selected neighbors info
and put it into variables*/
nbcount=countnb; nbcases=&criteriavar.;
/*********************************************************
/*calculate information about the new region                *
*********************************************************/
newcases=firstcases+&criteriavar.;
/*geographic centriod of new area*/
/*could change this to be population-weighted, if desired*/
   newlat=(firstlat+lat)/2;
   newlon=(firstlon+lon)/2;
/*if the new region is within a certain block grp, tract, or county assign it
that block grp, tract, or county*/

newcounty=nbcounty; /*assign the new region to a county*/
*get the new count of neighbors of new region;
declare hash nl();
nl.definekey('id');
nl.definedata('id');
```

```
nl.definedone();
declare hiter rnl('nl');
*use hash object nl to hold list of neighbors to determine count;
do x=1 to sum(of firstcountnb,nbcount);
   rc1=nb.find(key:firstid, key:x);
   if rc1=0 and (neighbor NE nbid) then rc=nl.add(key:neighbor, data:neighbor);
   rc2=nb.find(key:nbid, key: x);
   if rc2=0 and (neighbor NE firstid) then rc=nl.add(key: neighbor,
   data:neighbor); /*with add method, duplicate keys should be ignored*/
end;
newcountnb=nl.num_items;
/*************************************************************
/*end section which calculates information about the new region *
*************************************************************/
```

Once the new region is created, it will become a candidate to become the next input area. It's information must be added to the hash objects 'blist' and 'ord' using the ADD method. The old input areas, which will no longer exist, must be removed from 'blist' and 'ord'. To do this, we must first remove their hash iterators using the DELETE method; this 'unlocks' the hash objects so the REMOVE method can be used to get rid of the information about the old input areas. Lastly, and most importantly, we create a record of the geographic merge in the 'idregion' hash object by using the ADD method to create two records, each consisting of an input area id and the output region id.

```
/*add new data to blist and ord */
rc=blist.add(key:charreg,
   data:charreg, data:newcases, data:newcounty, data:newlat, data:newlon,
data:newcountnb);
 rc=ord.add(key:newcases,  key:newcountnb, key:charreg,
          data:newcases, data:newcountnb, data:charreg);
/*remove old data from blist and ord*/
/*we must delete and recreate the iterator before removing data or errors
          will be created if this code is run in SAS 9.2*/
 rc=rblist.delete();
 rc=blist.remove(key: firstid); rc=blist.remove(key: nbid);
 rblist=_new_ hiter('blist');
 rc=rord.delete();
 rc=ord.remove( key:firstcases,   key:firstcountnb, key:firstid);
 rc=ord.remove( key:nbcases, key:nbcount, key: nbid);
 rord=_new_ hiter('ord');
/*add the input areas and output region to the output hash object*/
 rc=idregion.add(key:nbid, data: nbid, data:charreg);
 rc=idregion.add(key:firstid, data:firstid, data:charreg);
```

After the new region is created, the contiguity information in the contiguity table (hash object 'nb') must be updated to reflect the new geographical relationships. The new output region must be added, and the old input areas must be removed. The records pertaining to the neighbors of the new output region must be updated to reflect that they are no longer neighbors of the input areas. If the output region's number of neighbors has changed (usually because it borders on both input regions), then the 'blist' and 'ord' hash objects must be updated to reflect this.

```
*************************************************************
*update the contiguity hash object                          *
*************************************************************;
do x=1 to newcountnb; /*add region with its list of neighbors*/
   if x>1 then rc=rnl.next(); else rc=rnl.first();
   /*add the new region and its neighbors to the contiguity hash object*/
   rc=nb.add(key:charreg, key: x, data: charreg, data:x, data:id); regnbid=id;
   /*update the entries in the contiguity hash object for the neighbors of
   the new region so they show that they border the output region,
   not the input areas*/
   rc=blist.find(key:regnbid); /*find countnb for this neighbor*/ rflag=0; z=0;
   do y=1 to countnb;
```

```
            rc=nb.find(key:regnbid, key:y);
            if rc=0 then do; z=z+1; /*z is a count of the new number of neighbors*/
                  if (neighbor=firstid or neighbor=nbid) and rflag=0 then do;
                  /*using the replace method to alter data which is also a key
                  gives funny results.  use remove & add instead*/
                        rc=nb.remove(key:regnbid, key: y);
                        rc=nb.remove(key:regnbid, key:z);
                        rc=nb.add(key:regnbid, key: z, data:regnbid, data:z, data:
                        charreg);
                        rflag=rflag+1;
                        /*if a neighbor borders both joined objects, there will be a
                        duplicate and rflag will become 1 or more*/
                  end;
                  else if (neighbor=firstid or neighbor=nbid) and rflag>0 then do;
                        rc=nb.remove(key:regnbid, key:y); z=z-1;
                  end;
                  else if y NE z then do;
                        /*do not use replace method for this - use remove and add*/
                        rc=nb.remove(key:regnbid, key: y); rc=nb.remove(key:regnbid,
                        key: z);
                        rc=nb.add(key:regnbid, key:z, data:regnbid, data: z,
                        data:neighbor);
                  end;
            end; *if rc=0 that is, if we find regions's neighbor's neighbor;
      end; *cycle through all neighbor's neighbors;
      /*update the number of neighbors for the neighbors; will only change if the
   neighbor borders both joined objects*/
      rc=blist.find(key:regnbid);
      if countnb NE z then do; /*need to update both for ord and blist*/
            rc=rblist.delete();
            rc=blist.replace(key:regnbid, data:regnbid, data: &criteriavar.,
            data:county, data:lat, data:lon, data:z);
            rblist=_new_ hiter('blist');
            rc=rord.delete();
            rc=ord.remove(key:&criteriavar.,  key:countnb, key:id);
            rc=ord.add( key:&criteriavar.,  key:z, key:id,
                        data:&criteriavar.,  data:z, data:id);
            rord= _new_ hiter('ord');
      end;
   end; /*go through all neighbors*/
   /*remove input areas which were aggregated from contiguity hash object*/
      do x=1 to max(firstcountnb, nbcount);
            rc1=nb.remove(key:firstid, key:x);
            rc2=nb.remove(key:nbid, key: x);
      end; *cycle through all input areas;
   ********************************************************************************
   *end update of contiguity hash object
   ********************************************************************************;
   end; *do until newcases>10000 or whatever minimum criteria is;
```

After merging two input areas into an output region, the output region is now a candidate to become an input area for a future merge. Merging continues until the criteria is met and the DO-UNTIL loop ends. After that, the hash objects that are no longer needed are removed using the DELETE method. This frees up computer memory for other uses and is good practice.

In order to be able to use the results to make a map, we must associate each original input area with the final output region to which it was assigned; for example, if original input area A was merged with B to form output region 1, and subsequently new input area 1 was merged with C to form output region 2 (which had sufficient population, and was not merged further) then original input area A must be assigned to output region 2, not to output region 1.

This is conveniently accomplished by using a SET statement to look through all the original input areas, and the FIND method in a DO-UNTIL loop to look through the output regions in hash object 'idregion' until the final output region is obtained. The first key for the FIND method is the id of the original area read in from the data in the SET statement. If this area has been merged, the method will return the id of the new region. The new region id is then used as the key for the FIND method, and if it has been merged again it will return an id for the new new region. This continues until no new region id's are found, and therefore the return code of the FIND method is non-zero. Thus, each original area ID is associated transitively with the final output region it is part of.

```
/*housekeeping: delete the hash objects, frees up the memory
they use for your computer to do other stuff*/
rc=nb.delete();
rc=ord.delete();
rc=blist.delete();
rc=nl.delete();
end; *if _n_=1 --- this is the end of loop in which all the aggregation
takes place*/
*match each input area with the final region it was assigned;
set allareas end=eof;
oldid=id; newregion=id; /*default is that the input area is not aggregated*/
do until (rc NE 0);
   rc=idregion.find(key:newregion);
end;
/*delete the last hash object when everything is finished*/
if  eof=1 then rc=idregion.delete();
run;
```

MAPPING THE RESULTS

The result is a data set containing the unique identification of each input area and the final output region to which it was assigned. This data set, along with the map data set, can be used to create a map in SAS. SAS supplies a wide variety of map data sets, but map data sets can also be created from Geographic Information Systems (GIS) data using the MAPIMPORT procedure. This procedure converts a shapefile (a common GIS format) to a SAS map data set. In this example, I import a shapefile obtained from New York State Office of Cyber Security & Critical Infrastructure Coordination (2007). The map data set and the aggregation data set are sorted and merged. The GREMOVE procedure (Zdeb, 2002) is then used to remove the internal boundaries within the aggregated areas. PROC MEANS is used to calculate statistics for the aggregated areas. Lastly, the GMAP procedure is used to create the map.

```
/*first, add region ids to original map*/
proc sort data=finalregions (rename=(oldid=&idvar.)); by &idvar.;run;
proc sort data=mymap;by &idvar.; run;

data step1map;
merge finalregions (keep=&idvar. newregion) mymap;
by &idvar.;run;

proc sort data=step1map; by newregion;run;
/*proc gremove joins the areas into regions*/
proc gremove data=step1map out=finalmap;
by newregion;
id &idvar.;
run; /*a few seconds for zip codes*/

/*summarize the data by region to map it*/
proc means data=finalregions noprint nway;
class newregion;
output out=finalmapdata sum(&criteriavar.)=&criteriavar.;
run;

/*create a chloropleth map of the new regions shaded by your variable*/
```

```
proc gmap map=finalmap data=finalmapdata;
id newregion;
choro &criteriavar.;
run;quit;
```

Figure 2 shows the map of input areas, and Figure 5 shows the map of output regions. Figure 4 shows the population distribution before and after merging.

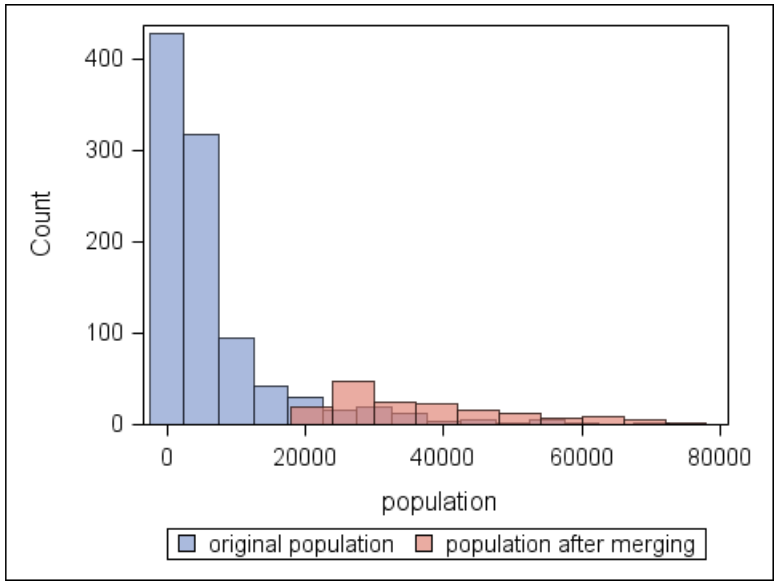

Figure 4: Histogram of input areas and output regions with a total population of less than or equal to 75,000. The input areas were the cities and towns of New York State and the output regions are combinations of the cities and towns that have a minimum population of 20,000.

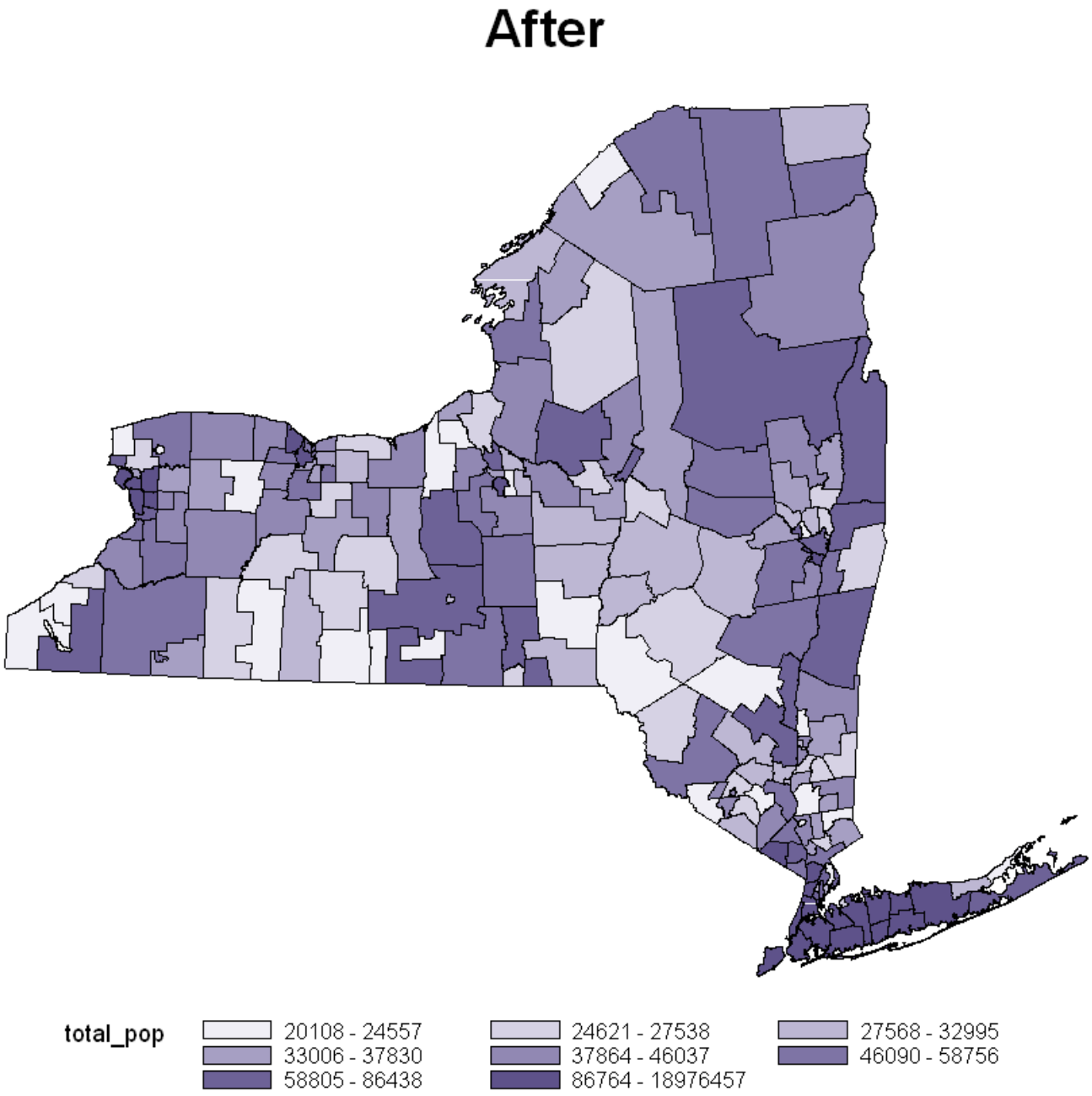

Figure 5: Cloropleth map of New York State regions created by merging towns and cities until regions have a minimum population of 20,000, as output from the SAS 9.2 GMAP procedure. This map was created with information copyrighted by the New York State Office of Cyber Security and Critical Infrastructure Coordination © 2007.

CONCLUSIONS

Using hash objects allows us to quickly join input areas into larger output regions. This program merged the 1013 cities and towns in New York State into 200 regions. Before merging, the population of the cities and towns in New York ranged from 0 (for 'towns' having only water in their boundaries) to 2,465,326 (Brooklyn), with a median of 3,183; after merging, the populations of the new regions ranged from 20,108 to 2,465,326, with a median of

37,830. Thanks to the use of hash objects, the merging takes only from a few seconds, as in this example, to a few minutes, for example for 290,000 census blocks in New York.

Using hash objects allows us to quickly compare a map data set to itself to determine which areas are neighbors, a task which would otherwise require sophisticated GIS software and considerably more time.

REFERENCES

- Dorfman, Paul M. (2008) "The DoW-Loop Unrolled" *Proceedings of the Northeast SAS Users Group*. <http://www.nesug.org/Proceedings/nesug08/hw/hw02.pdf>
- Dorfman, Paul M. and Shajenko Lessla S. (2007) "The DOW-loop unrolled" Paper SD08. *Proceedings of the SouthEast SAS Users Group Meeting*. <http://analytics.ncsu.edu/sesug/2007/SD08.pdf>
- Dorfman, P.M. And Vyverman, K. (2009) "The DOW-loop unrolled" *Proceedings of the SAS Global Forum*. <http://support.sas.com/resources/papers/proceedings09/038-2009.pdf>
- El Emam, Khaled; Brown, Ann; and AbdelMalik, Philip (2009). "Evaluating Predictors of Geographic Area Population Size Cutoffs to Manage Re-identification Risk" *J Am Med Inform Assoc*. 16(2):256-66.
- New York State Office of Cyber Security & Critical Infrastructure Coordination (NYS CSCIC) (2007) *CityTown* Albany NY. <http://www.nysgis.state.ny.us/gisdata/inventories/details.cfm?DSID=927> (July 31, 2007)
- SAS Institute Inc. (2008) *SAS/GRAPH® 9.2 Reference*. Cary, NC:SAS Institute Inc. p.708
- Zdeb, Mike (2002) *Maps Made Easy Using SAS®*. Cary, NC:SAS Institute Inc.
- Zdeb, Mike. (2004) *The Basics of Map Creation with SAS/GRAPH®*. Paper 251-29, SUGI 29.
- Ward, David. "TIP00108 - Use the Haversine formula to calculate distance between two points on earth using Latitudes/Longitudes" *SAS Coding Tips and Techniques* <http://www.sconsig.com/sastips/tip00108.htm> (June 30, 2009).

ACKNOWLEDGMENTS

Thanks to Steve Forand, Sanjaya Kumar, Thomas Talbot, the New York State Department of Health, Center for Environmental Health and the New York State Cancer Registry for supporting this work.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Gwen D LaSelva
New York State Department of Health
547 River St
Troy, NY 12180-2216
Work Phone: 518-402-7950
Fax: 518-402-7959
Email: gdb02@health.state.ny.us