更多数学建模资料获取

B 站数学建模试看课程：

https://www.bilibili.com/video/av20238704

清风数学建模视频课程的课表：

http://note.youdao.com/noteshare?id=2e1cc342928466bb632b33304a56e368

视频录制了 45 个小时，累计 800 多页 PPT，讲解的了 20 多个常用模型，内容非常详细，并配有作业讲解。

关注微信公众号：《数学建模学习交流》可获得更多数学建模技巧和比赛信息。

怎么在手机上购买资料：在公众号后台发送“买”这个字，就可以获得一个链接，点进去后就可以购买了（购买方式和淘宝一样）

What to Feed a Gerrymander

Control # 1421

Mathematics Contest in Modeling
February 12, 2007

Abstract

Gerrymandering, the practice of dividing political districts into winding and unfair geometries, has a deleterious effect on democratic accountability and participation. Incumbent politicians have an incentive to create districts to their advantage (California in 2000, Texas in 2003) so one proposed remedy for gerrymandering is to adopt an objective, possibly computerized, methodology for districting. We present two computationally efficient algorithms for solving the districting problem by modeling it as a Markov decision process rewarding traditional measures of district "goodness": equality of population, continuity, preservation of county lines, and compactness of shape. Our *Multi-Seeded Growth Model* simulates the creation of a fixed number of districts for an arbitrary geography by "planting seeds" for districts and specifying particular growth rules. The result of this process is refined immensely in our *Partition Optimization Model* which uses stochastic domain hill-climbing to make small changes in district lines that improve goodness. We include, as an extension, an optimization to minimize projected inequality in district populations between redistrictings. As a case study, we implement our models to create an unbiased, geographically simple districting of New York using tract-level data from the 2000 Census. We conclude with an open letter to members of the New York State Assembly.

What to Feed a Gerrymander

Team 1421

1 What is Gerrymandering?

Gerrymandering is the division of an area into political districts that give special advantages to one group. Frequently, this involves concentrating "unfavorable" voters in a few districts to ensure that "favorable" voters will win in many more districts. In order to squeeze all of the unfavorable voters into a few districts, gerrymandering creates snaky and odd shaped regions. The eponymous label was created when politician Elbridge Gerry pioneered this technique in early 19th Century and his opponents claimed the districts resembled salamanders.

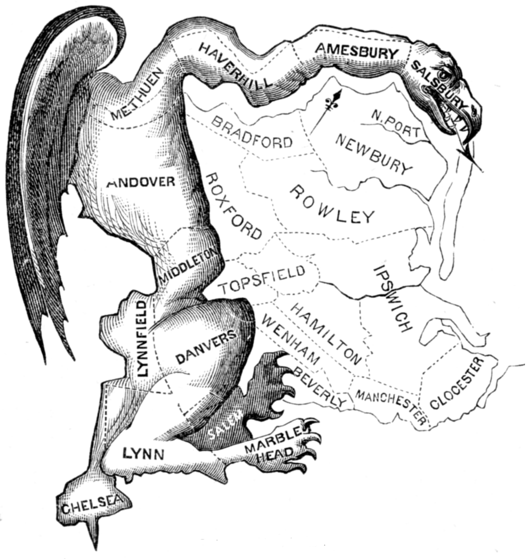


Figure 1: The original "Gerry-mander" from the *Boston Centinel* (1812)

1.1 Basic Terminology

- **Packing** - Forcing a disproportionately high concentration of a particular group into one district to lessen their impact in nearby districts.
- **Cracking** - Spreading out members of some group in several districts in order to reduce their impact in each of these districts.
- **Forfeit district** - A district where group A packs the members of group B so that group B wins this district but loses several surrounding districts which B may have won with a different districting scheme.

- **Wasted Vote** - A vote cast by a member of group A in a district where A is already assured victory so voting has no bearing on the result. In general, the group with more wasted votes is made worse off by a districting plan.

1.2 Why is it so bad?

Politicians today still gerrymander federal and state-level electoral districts and the public outcry is still strongly negative. Before we set out to eliminate this practice we should discuss why gerrymandering is considered problematic.

First off, gerrymandering reduces electoral competition within districts since cracking/-packing makes elections uncompetitive. Further, incumbent representatives are in no real danger of losing elections so they do not campaign vigorously which can lead to lower voter turnout. Exacerbating the problem, incumbents' increased advantage means they are less incentivized to govern based on their constituents' interests so democratic accountability and engagement mutually deteriorate.

Gerrymandering also presents the practical problem that it is difficult to explain to voters why district shapes are so labyrinthine. Some districts connect demographically similar but geographically distant regions using thin filaments such as the district depicted in Figure 2. "Niceness" of district shape almost always takes a back seat to political and racial concerns when districts are being created. Example: In the 2000 California realignment, Democrats and Republicans united to design incumbent-favoring districts which resulted in the reelection of all of the 153 involved legislators in 2004. How can one argue that this is in voters' best interests?

However, it should be noted that gerrymandering can be considered appropriate in specific situations. For instance, the Arizona Legislature gerrymandered a division between the historically hostile Hopi and Navajo tribes even though the Hopi reservation is entirely surrounded by the Navajo reservation.

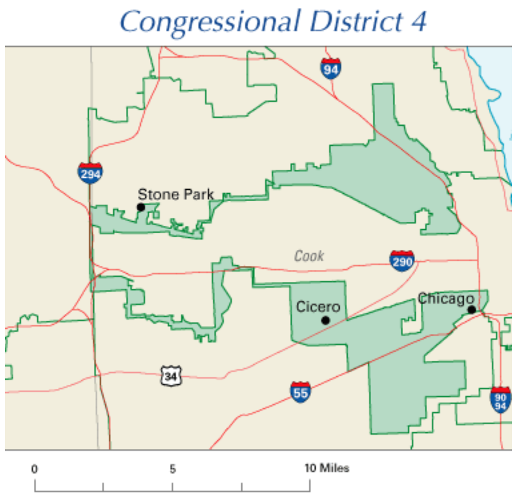


Figure 2: A present-day gerrymander, the Illinois 4th congressional district. (The two "earmuffs" are connected by a narrow band along Highway 294.)

1.3 The legality of gerrymandering

We should be clear on one point: though gerrymandering is objectionable to many, it is legal around the country. Interestingly, the Voting Rights Act of 1965 which eliminated poll taxes and other discriminatory voting policies may have inadvertently increased the prevalence of gerrymandering. One interpretation of the Act was that it mandated nondiscriminatory election *results* which led to a strange reversal of vocabulary where creating "majority-minority" districts was considered beneficial. These gerrymandered districts were packed with minorities which guaranteed minority representation in Congress.

However, in Shaw v. Reno (1993), and later in Miller v. Johnson (1995), the Supreme Court ruled that racial/ethnic gerrymanders were unconstitutional. Nevertheless, Hunt v. Cromatrie approved of a seemingly racial gerrymandering since the motivation was mostly partisan rather than racial. The recent case League of United Latin American Citizens v. Perry (June 2006) upheld the position that states are free to redistrict as often as they like so long as these redistrictings follow are not purely racially motivated.

2 Assumptions and Notation

2.1 What can we consider when districting?

We have compiled the following list of possible factors one might consider is districting a State. The list is ranked with factors we consider more important or legitimate at the top.

1. Population equality between districts (legally mandated)
2. Continuity of districts (legally mandated, excepting islands)
3. Respect for legal boundaries (counties, city limits, townships)
4. Respect for natural geographic boundaries
5. Compactness of district shapes
6. Respect for man-made boundaries (highways, parks, etc.)
7. Respect for socio-economic similarity of constituents
8. Similarity to past district boundaries
9. Partisan political concerns
10. Desire to make districts (un)competitive
11. Racial/ethnic concerns
12. Desire to protect (or unseat) incumbent politicians

We consider only the top seven factors in our model. Factors 9-12 are all related to political or racial concerns which our model is specifically designed to ignore. The case SC State Conference of Branches v. Riley (1982) ruled that past districts (Factor 8) are a legitimate tool for creating new districts but we choose to ignore past districts since they are heavily biased by Factors 9-12.

2.2 Geography and similar characteristics

The US Census Bureau provides a great deal of data on legal, natural, and man-made boundaries as well as socio-economic similarity of regions. In each census, the United States is broken up into several degrees of accuracy, the smallest of which are: blocks (40 people on average), block groups (1500 people), and tracts (4500 people).

We follow the practice in Young (1988) by districting based on a maximum level of resolution which in our Case Study (Section 5) is census tracts. Notational note: we refer to the smallest unit of division generally as a *tract*.

A reference from the Caliper Corporation describes tracts in the following quotation:

> Census tract boundaries normally follow visible features, but may follow governmental unit boundaries and other non-visible features, and they always nest within counties. Census tracts are designed to be relatively homogenous units with respect to population characteristics, economic status, and living conditions at the time the users established them.

For these reasons we believe that units at the tracts size (or less) are acceptably small and homogenous to use as a base unit. Further, tracts are completely contained within counties so we can easily check whether or not a district breaks county integrity.

2.3 Notation

Define m to be the number of census tracts, and n the number of districts.

We denote our districts by $D_i, 1 \leq j \leq n$, and our tracts by $T_l, 1 \leq l \leq m$. Denote the set of all tracts by $\Gamma = \{T_l\}_{1 \leq l \leq m}$; we call this a *State*. Denote the set of all districts at a particular time by $\Delta = \{D_i\}_{1 \leq j \leq n}$. We call this a *partition* for the State.

2.3.1 Adjacency

Define the symmetric relation $T_p \sim T_q$ for tract pairs (T_p, T_q) which are adjacent. Define the function $d(T_l)$ to be the district to which the tract T_l belongs. We also naturally extend the definition of d to sets of tracts.

Define the *neighbor set* of tract T_l by $a_T(T_l) = \{T_p \in \Gamma | T_l \sim T_p\}$ to be the set of all census tracts neighboring T_l, and define $a_D(T_l) = d(a_T(T_l))$ to be the set of all districts containing neighbors of T_l. Every tract borders at least one other tracts, so $a_T(T_l)$ and $a_D(T_l)$ have cardinality at least one for all T_l.

2.3.2 Borders

Define the *border* of district D_i as $\partial D_i = \{T_l \in D_i | a_D(T_l) \neq \{D_i\}\}$ which is the set tracts in D_i that are adjacent to at least one district other than D_i. The *interior* of district D_i is $I_i = D_i \backslash \partial D_i$, the set of census tracts in D_i whose neighbors are all in D_i. Denote the total number of tracts in district D_i as $m_i = |D_i|$ the number of border tracts as $b_i = |\partial D_i|$.

The *frontier* of D_i is denoted $F_i = (\cup_{T_l \in D_i} a_T(T_l)) \backslash D_i$, i.e. the set of all tracts outside of D_i that border the boundry tracts of D_i.

2.3.3 Counties

We denote a county as C_j and the set of all counties as Λ. Districts can (and often do) break county boundaries but tracts are contained entirely within counties so we can think of a county as a set of districts. Districts are also sets of tracts so we interpret the set intersection $D_i \cap C_j$ as the set of tracts in both district D_i and county C_j. From this, we define $c(D_i) = \{C_j | D_i \cap C_j \neq \emptyset\}$ to be the set of counties which overlap with D_i.

2.3.4 Population

Let the population of our State be P and we denote the optimal district size, $\frac{P}{n}$, as $\bar{p}$. We use the function $p(\cdot)$ to generally denote the population of an object, for instance $p(T_l)$ and $p(C_j)$ are the populations of tract T_l and county C_j, respectively. Due to frequent use, we use the shorthand $p_i = p(D_i)$ for the population of districts.

Table 1 is a useful reference of these numerous definitions.

Table 1: Variables and their meanings

Variable	Definition
n	Number of congressional districts
D_i	The ith district ($1 \leq i \leq n$)
Δ	Set of all districts in a State, a *partition*
m	Number of census tracts
T_l	The lth tractfin ($1 \leq l \leq m$)
Γ	Set of all tracts in a State
$d(T_l)$	District to which tract T_l belongs
$T_p \sim T_q$	Tracts T_p and T_q are adjacent
$a_T(T_l)$	Set of tracts adjacent to tract T_l
$a_D(T_l)$	Set of districts containing tracts neighboring T_l
∂D_i	Border of D_i, tracts that neighbor another district
I_i	Interior of D_i, tracts with do not neighbor another district
m_i	Number of tracts in D_i
b_i	Number of tracts in ∂D_i
F_i	Set of all tracts outside of D_i that border ∂D_i
C_j	The jth county
$c(T_l)$	The county to which tract T_l belongs
$c(D_i)$	The set of counties containing district D_i
P	Total population of the State
$\bar{p}$	Average population of a district
$p(\cdot)$	Population of an arbitrary object
p_i	Shorthand for $p(D_i)$, population of district D_i

2.4 Past Models

Prior to explaining our modeling approach we would like discuss some previous work in the literature on congressional districting and gerrymandering. We used these papers as guides as we thought about and further refined our algorithm and implementation.

Cirincione *et al.* (2000) judge the quality of a districting plan based on equal population, preservation of county integrity, and district area compactness. They require that district populations differ by no more than 1% from exact equality in the number of constituents, and require point contiguity of the districts. The algorithm constructs districts by picking a random block group (their unit size), then adding additional block groups to the new district until the district population reaches $\bar{p}$. At this point they repeat the process starting with a new random block group. Compactness of districts is based on their minimum bounding rectangles and county integrety is encouraged by "randomly" selecting new block groups with a preference for block groups in already inhabited counties.

Mehrotra *et al.* (1998) and Garfinkel and Nemhauser (1970) implement a "branch-and-price" method in the optimization step. They first obtain a districting, and optimize over their constraints such that population values are allowed to vary in the final solution of the optimization step. In a final step they split up population units to ensure population equality. They define compactness in a graph-theoretical manner where connected nodes are adjacent tracts. They define the "center" of a district to be the node (tract) with the lowest maximum distance to another other tract. They consider a graph (district) more compact when sum of distances from each node to the center is small.

We do not use this measure, as it does not uniquely define the center of a graph, and, contrary to their claims, does allow for oddly-shaped districts, such as a district whose graph is a star-shaped tree with one tract in the center and many non-contiguous paths emanating from it. In our case study simulations, prior to the incorporation of a compactness factor in the objective function, we often obtain such a tree structure, which is one of the salient features of gerrymandering.

We also do not use a "branch-and-price" method of optimization. Following suggestions of Nagel (1965) and Kaiser (1966), we employ a local search algorithm in which tracts are swapped between existing districts to maximize some objective function. We describe this process in detail in Section 4.

2.5 Measuring compactness

The notion of compactness of a planar region has no uniformly accepted definition and research done by Young (1988) suggests that any reasonable measure of compactness fails to work well for certain geographic configurations. He further suggests that any good measure of compactness in such problems should consider the population units (census tracts in our case study) as indivisible units, and therefore that the measure of compactness should be made independently of the predetermined shapes of the population units. We follow this directive in our definition of compactness.

In fact, the compactness measures given in Young (1998) are not reasonable in the first instance, and do not include any notion of the area of a district, or comparing it to the perimeter. The measures include the maximum total perimeter of a district in a districting, determining the relative height and width of the district, and finding the moment of inertia of the district. All of these measures fail to consider both perimeter and area simultaneously, which seems to be a reasonable requirement of a good compactness measure.

The Isoperimetric Theorem, first proved (non-rigorously) by J. Steiner in 1838, states that the quantity A/P^2, given by the ratio of the area A of a planar region (not necessarily continuous) to the square of its perimeter is maximized when the region is circular. The maximum achievable compactness, that of a circle with radius r, is given by $\frac{\pi r^2}{4\pi^2 r^2} = \frac{1}{4\pi}$ so we define *compactness* of a region as the ratio $(4\pi A)/P^2$. This ratio is bounded within $(0, 1]$ where higher values indicate greater compactness.

We believe this serves as a good measure of the broadly defined "regularity" of a region which is so important to the study of Congressional districting and gerrymandering. Specifically, any shear of factor s applied to a circle decreases the compactness by a factor of s, and any concave region has lower compactness than does its convex hull. It is easy to see that we can make an even stronger statement: the convex hull of a concave region has greater area *and* smaller perimeter.

Observe that a square gets close to the optimum, with a compactness of $\frac{4\pi}{16} \approx 0.785$. This implies that the set of possible compactness values for rectangles is $(0, 0.785)$ since a square is the most compact rectangle.

3 The *Multi-Seeded Growth Model*

We take a two-stage approach to finding the best districts for a given State. In the *Multi-Seeded Growth Model*, referred to as *MSGM* hereafter, we find an initial allocation of n districts so that the partition has modest levels of population equality and county preservation. Our more precise *Partition Optimization Model*, or *POM*, edits and improves the rough sketch from *MSGM* into until it becomes, hopefully, a work of art.

The reason that our model runs in two phases is simple: speed. Our knee-jerk reaction to the problem was to randomly allocate tracts to the n districts and then optimize by swapping tracts trying to improve some objective function. However, a random initial configuration is so far from the global maximum that the search might take millions of years.

The *MSGM* generates a very crude coloring of a State that ensures district contiguity and tries, but does not guarantee, to achieve population equality and county preservation. The districts created by *MSGM* are completely unacceptable for an actual plan but save enormous amounts of computing time for our solution.

3.1 How it works

At first, our task seems daunting. How do we allocate n districts equally, even to a rough approximation? Our solution is to grow the n districts simultaneously until they cover the State.

We start by allocating the entire State to a blank, dummy district D_0, and then allocating n tracts that serve as the initial "seeds" for the final districts, such that each $D_i, i \in \{1, \ldots, n\}$ begins as only a single tract. Now while $|D_0| > 0$, we take the set S of all possible moves which involve taking a district from D_0 while preserving contiguity. That is:

$$S = \bigcup_{i=1}^{n} \bigcup_{T_l \in F_i} M(T_l, D_0, D_i)$$

Where $M(T_l, D_i, D_j)$ represents a *move of tract T_l from D_i to D_j*, corresponding to the exit of T_l from D_i and the entrance of T_l into D_j. We then sort the moves in S by our heuristic function $\Psi(D_1, \ldots, D_n) \to \mathbb{R}$, a function increasing in the desirability of our prospective partition. Each move is scored by the heuristic value that would result if we were to accept only that move. We then conclude by performing the moves corresponding to the top 3% of the scored moves in S. Note that this method preserves contiguity, because by definition any $T_l \in F_i$ must be contiguous with D_i, and thus the D_i are contiguous at each step.

Had we but world enough, and time, we would only perform the best possible move found in S before recalculating the frontier. Even though in the *MSGM* we do not consider moves between two "true" districts (rather, we consider only moves between a true district and the dummy district), the value of a move does not exist in isolation. Consider two distinct districts D_i and D_j, and two tracts $T_l \in F_i \cap F_j$ and $T_k \in F_i \cap F_j^c$. The acceptance of $M(T_k, D_0, D_i)$ alters the heuristic value of every move associated with F_i, which could potentially affect the optimality of further moves with D_i, such as the acceptance of $M(T_l, D_0, D_i)$ rather than $M(T_l, D_0, D_j)$. Furthermore, the acceptance of $M(T_l, D_0, D_i)$ likely expands the size of F_i. Perhaps there is an optimal move opened up in this new frontier that we do not even consider, because we have not even calculated its value.

It would be better to only perform the best move, but such a strategy was found to be too computationally intensive. We compromise by taking only a small, elite fraction of the moves in each step before recalculating S and the values of its associated moves. In this respect, our approach is analogous to the strategy of *modified policy iteration* for solving a Markov decision problem. And just as modified policy iteration excels in practice, we found that the tradeoff of possible inefficiency is more than compensated for by the speed gains of the algorithm, especially considering that the solution obtained by *MSGM* will be further refined by *POM*.

In true modified policy iteration, k rounds of value iteration are made in-between policy iterations, such that k is fixed. Our *MSGM* scheme uses a variable number of moves in-between recalculating the value of the frontier. We selected our scheme because it causes us to be delicate in our selections of tract allocations, making moves virtually one-at-a-time, at the beginning and end of the *MSGM*. By focusing on the beginning and end of the problem, we attempt to avoid having a single district grow too large through possible inefficient allocation.

Unlike Cirincione (2000) we use random initial seeds weighted by population rather than seeds that are equally spaced around the State. The process works as follows: while there are still random seeds to be selected, we find a candidate initial seed tract T_l in D_0. Letting the largest tract in our State have population $\hat{p}$, we accept T_l as an initial seed with probability $p(T_l)/\hat{p}$. We thus select tracts in linear proportion to their population. We found that the *MSGM* algorithm produces the best initial results when all the districts have the same amount of *population*, rather than the same *number of tracts* around which to grow. The geographically optimal placement of five, or fewer, starting seeds in the NYC Metropolitan area and Long Island evinces the fallibility of the equidistant initial seed method.

We have presented our scheme for growing emerging districts, but we should also discuss the heuristic by which we rank candidate moves. It has two components: a population score and a county score.

3.2 Population score

Even thought the *MSGM* is only an rough start for our optimization we would like to minimize egregious disparities in population between districts. We would much prefer if the *MSGM* produces a result where the largest district has twice the population of the smallest rather than 100 times the population.

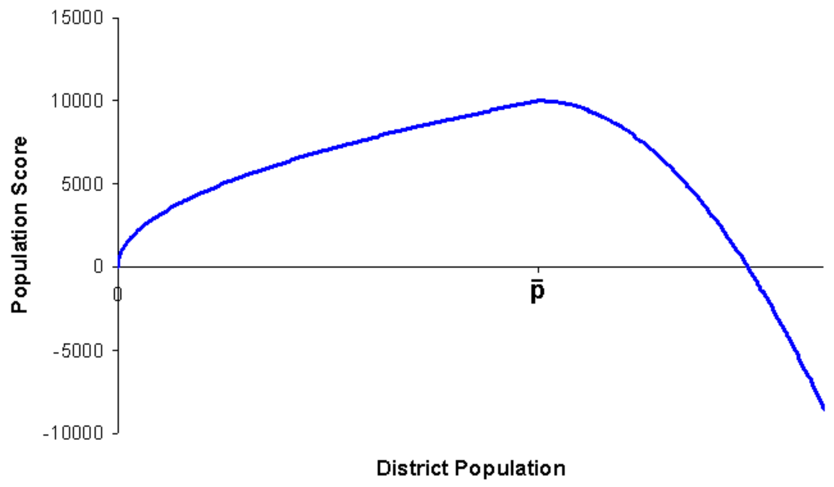
Clearly, the population component of our heuristic should give the highest score to a district when $p_i = \bar{p}$. Additionally we want to penalize large deviations from the optimal population level so our function should be concave down.

Admittedly, choosing a heuristic is somewhat arbitrary but this does not bother us since the results from *MSGM* are only a baseline. Let $f(p_i)$ be the population heuristic score for a district with population D_i. We use a piecewise definition for f:

$$f(p_i) = \begin{cases} M\sqrt{\frac{p_i}{\bar{p}}}, & \text{if} p_i \leq \bar{p} \\ M - \frac{4M}{p_i^2}(p_i - \bar{p})^2, & \text{if} p_i > \bar{p} \end{cases} \tag{1}$$

Notice that f is steeper for values $p_i > \bar{p}$ because we do not want *growing* districts to engulf too much population; we penalize deviations above $\bar{p}$ worse than deviations below $\bar{p}$. (We also consider some "nicer" functions, like a Beta distributions, but we opted for a computationally simpler implementation.) Figure 3 shows the function f.

Figure 3: *MSGM* heuristic for population



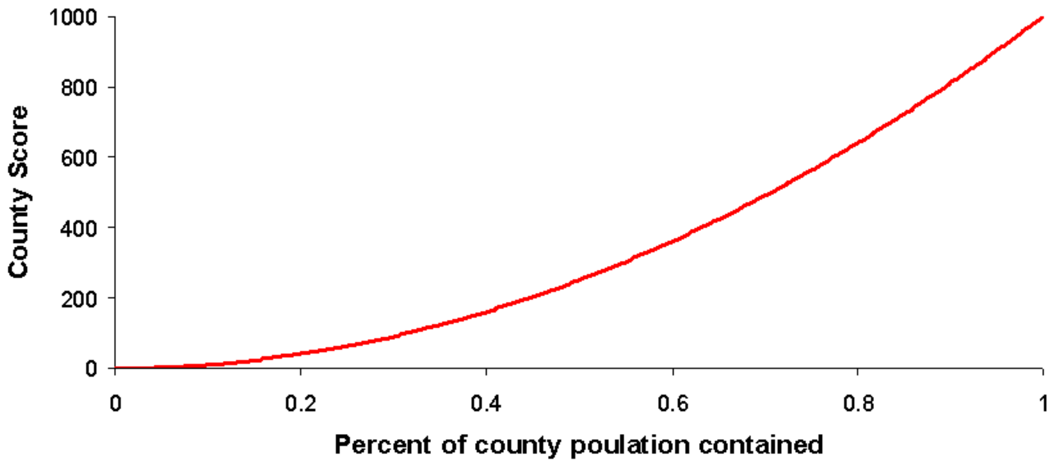
3.3 County preservation score

For a given district D_i, we measure its county preservation score in terms of the percent of counties that it completes on a population basis. To encourage growing districts to add remaining tracts in nearly complete counties the marginal value adding these should increase with the fraction of the population already contained in that district. To accomplish this we use the square of the proportion contained in a county. The county score, g, for a district D_i is:

$$g(D_i) = \sum_{C_j \in \Lambda} \left(\frac{\sum_{T_l \in D_i \cap C_j} p(T_l)}{p(C_j)} \right)^2 \tag{2}$$

For instance, if a district completely contains one county and contains 30% of each of two other counties' populations then its score would be $(1^2 + .3^2 + .3^2) = 1.18$. Figure 4 shows a plot of the county score a district receives based on what percent of a counties population said district contains.

Figure 4: *MSGM* heuristic for county completeness



4 The *Partition Optimization Model*

Now that we have constructed a crude, approximate solution to the districting problem by using *MSGM*, we refine the solution through a process of local search. We define our local search by our objective function, and our neighborhood function and search space.

4.1 The objective function

For our optimization function, the only characteristics of each district and each county we will use are the populations $p(P) = \{p_i\}_{1 \le i \le n}$, the compactness measures $c(P) = \{c_i\}_{1 \le i \le n}$, and the fractions $\rho(P) = \{\rho_{i,r} | 1 \le i \le n, 1 \le r \le c\}$ of the population of county r which is contained in district i. Based on our analysis of desired properties of districts, we would like our score function $s(P) = s(p(P), c(P), \rho(P))$ to have the following properties:

1. the score function should be unimodal as a function of p_i, with mode at $p_i = \bar{p}$;
2. The score should increase more by adding tracts which lie in $\chi(D_i)$, so that we prefer having as few districts as possible in a given county.
3. The score should increase by more by adding tracts which increase the sum of all compactness measures by the greatest amount.

When considering these three constraints, they suggest that we should consider the three vectors $p(P), c(P), \rho(P)$ independently of each other in the score function, and then compare the scores of each when deciding on how to make tradeoffs between population equality, compactness, and county unity. In other words, we would like our score function to be a separable function of these three vectors, i.e. s has the form

$$s(P) = f(p(P)) + g(c(P)) + h(\rho(P))$$

where f, g, h are functions.

4.1.1 One (wo)man, one vote

Based upon the first criterion, we only require a globally concave down function whose maximum is attained at $p_i = \bar{p}$ for all p_i: $\frac{\partial s}{\partial p_i}|_{p_i=\bar{p}} = 0$, $\frac{\partial 2s}{\partial p_i 2} < 0$. The simplest functional form which satisfies this constraint is:

$$f(p(P)) = -\alpha_p \sum_{i=1}^{n} (p_i - \bar{p})^2$$

where α_p is some constant. That is, the score attributable to population differences is actually a constant multiple of the population variance across districts (once all tracts are assigned to a district).

The *MSGM* creates districts with approximate population equality by penalizing extreme variation away from $\bar{p}$ but equality is generally pretty weak. In one, more or less typical run of *MSGM* the districts created vary from 600,000 to 700,000, an unacceptable difference for a final districting plan.

By far, the most important constraint in determining district lines is that the populations within each district are very similar. Note that, this criterion is based on the *general population* within districts not the voting-age population or the population of likely voters.

Recall that our State has total population P and an average population of $\bar{p} = P/n$ per district. Letting p_i be the population in district i we consider three potential metrics for the population variance between districts.

1. Variance: $Var(p_1, p_2, \ldots, p_n)$
2. Maximum deviation: $\max\{|p_i - \bar{p}|\}$

3. Maximum difference: $\max\{p_i\} - \min\{p_i\}$

For all of these measures lower values are preferable and the minimum value is 0. We submit that choice number 1, variance, is the superior alternative. To see why, consider two possible population distributions between districts:

- *Situation A* - one district has a population of $1.05\bar{p}$, one is $.95\bar{p}$, and all of the others are $\bar{p}$
- *Situation B* - half of the districts have population $1.05\bar{p}$ and half have $.95\bar{p}$ (any left over odd district has $\bar{p}$)

In *Situation A* only two districts are different from the ideal population level, $\bar{p}$, but in *Situation B* very few districts have population $\bar{p}$ so a good metric should rank B worse than A. Clearly, the variance of populations is higher in B than in A, so variance passes this test. The maximum difference test gives $.05\bar{p}$ for both A and B and the maximum difference gives $.1\bar{p}$ for both.

We see that variance is the best measure of similarity since it factors in the pair wise difference in all district populations. We use variance as our measure of populational inequality between districts.

4.1.2 Compactness

To measure the compactness of a district we would ideally use our compactness measure:

$$c_i = \frac{Area(D_i)}{Perimeter(D_i)^2}$$

Such that:

$$g(c(P)) = \beta \sum_{i=1}^{n} c_i$$

where β is some constant.

Unfortunately, try as we might, we were unable to calculate the perimeter of tracts on the aggregate - the C++ library we used to interact with our census data shapefiles exhibited a variety of disturbing characteristics for different methods we used for calculating perimeters, including massive memory leaks for large-scale union operations, questionable accuracy for pairwise unions, and seemingly arbitrary calculations of intersection length.

Yet it is a poor craftsman that blames his tools and so undaunted, we adopted a different measure of compactness. Called the *clustering coefficient*, it provides a rough approximation for compactness. We define it as:

$$cc(D_i) = \frac{\sum_{T_l \in D_i} |\{T_k \in D_i | T_k \sim T_j\}|}{\binom{m_i}{2}}$$

such that:

$$g(c(P)) = \beta \sum_{i=1}^{n} cc(D_i)$$

where β is some constant. Our clustering coefficient thus provides a ratio of the total number of inter-district boundaries to the maximum possible number of inter-district boundaries. Note that if all tracts were uniformly shaped, this measure would prize square- and circle-shaped districts, while winding, single tract-width districts would be penalized. However, given the asymmetry of tract shapes, this measure does little to reflect negatively upon district shapes such as the dumbbell, two circular clusters of tracts connected by a narrow band of tracts. In general however, the clustering coefficient will value adding to districts tracts that are "close" and removing from districts those tracts that are auxiliary.

4.1.3 County preservation

We adopt the same county preservation measure used in the *MSGM*, defined in equation 2 with the option of adding a scaling factor to the entire function to refine empirical performance.

4.2 Search method and neighborhood function

In order to refine our solution from *MSGM*, we must move tracts between districts. Yet the space of all possible contiguous moves is too large to run effectively. We solve this problem considering a range of possible moves with respect to only one district, its boundary and frontier, and performing the best move on this dramatically reduced state space.

By selecting our target district at random at each iteration, our strategy is best described as *stochastic domain hill climbing*. It is a method that combines the best aspects of both random and deterministic local search methods - we perform optimal moves while avoiding getting stuck trying to only increase the score of a single district. After determining that simple first-order moves on the district level, that is, adding or removing individual tracts, were incapable of reducing our variance metric to the extremely low standard that was our charge, we expanded our search to include second-order moves, that is, "swaps", a combined move that includes both an add and remove within a single operation.

If we assume that the maximum connectedness of any tract on the graph is k, checking for all adds and removes separately for district D_i involves considering $O(k|\partial D_i| + |F_i|) = O(km_i)$ possible moves, while looking at all swaps involves considering $O(k|\partial D_i||F_i|) = O(km_i^2)$ possible moves. One might contend, then, that the operation of checking *every* district for first-order moves might be a better algorithm, as it would take $O(\sum_{i=1}^{n} km_i) = O(nkm_i)$ heuristic evaluations. One could even supplement such an algorithm with a degree of randomness, to avoid being caught in a possible loop of futility, by employing simulated annealing, stochastic hill climbing, or tabu search on the resulting list of possible future states. In practice, however, we found that checking for second-order moves provided far better empirical results with acceptable time performance, while an algorithm enumerating all the possible second-order states, requiring $O(\sum_{i=1}^{n} km_i^2) = O(nkm_i^2)$ heuristic evaluations, was too slow to be effective.

The true heart of *POM* is the following algorithm. For simplicity and readability, we let $M_{add}(D_i)$ be the set of all moves in which we add a frontier tract to D_i, and $M_{remove}(D_i)$ to be the set of all moves in which we remove a border tract from D_i, and M^{-1} the move that is the inverse of M, such that applying both M and M^{-1} in turn has no effect. Recall also that our heuristic scores partition P as $s(P)$.

```
Input: Iteration count $iter$, initial partition $P$.
Output: Final partition $P$.
$count \leftarrow 0$
while $count < iter$ do
    $curscore \leftarrow s(P)$
    $D \leftarrow randomDistrict()$
    $bestscore \leftarrow curscore$
    foreach $M_a \in \{\emptyset \cup M_{add}(D)\}$ do
        foreach $M_r \in \{\emptyset \cup M_{remove}(D)\}$ do
            $performMove(M_a)$
            $performMove(M_r)$
            if $isContiguous(P)$ then
                $tmpscore \leftarrow s(P)$
                if $tmpscore > bestscore$ then
                    $bestscore \leftarrow tmpscore$
                    $bestadd \leftarrow M_a$
                    $bestremove \leftarrow M_r$
                end
            end
            $performMove(M_a^{-1})$
            $performMove(M_r^{-1})$
        end
    end
    if $bestscore > curscore$ then
        $performMove(bestadd)$
        $performMove(bestremove)$
    end
    $count \leftarrow count + 1$
end
return $P$
```

Algorithm 1 - Stochastic domain hill-climbing algorithm for districting

Note that we guarantee that our solution will be contiguous by not even considering moves that would break contiguity, and that we only perform a move if it increases the score of our current state.

4.3 Achieving absolute equality

US law mandates that the populations of each district be equal within a range of error of one person according to the census data (Karcher v. Daggett, 1983). Our problem dealt only with census tracts, and so exact equality of populations to the nearest integer was not possible. This last step of the algorithm must be implemented by splitting tracts between two districts.

To the knowledge of the authors, this problem beyond population unit level (no smaller than block groups) has not been addressed in the literature. Clearly, the simplest way to do this is to split one of the border tracts. While we do not implement this part of the algorithm in the computer simulation, we describe the methodology for doing this.

Let G denote the graph whose vertices are given by the districts and whose edges are the pairs of bordering districts. The intuition for the algorithm is that if we can find a pair of districts such that splitting a border tract between them gives both districts a population of one within the mean population, then we would optimally do so and ignore those two districts for the remainder of the algorithm. However, to guarantee that the algorithm finishes, we require that the graph G remain connected (otherwise, G may divide into two or more connected components, such that the constituent districts cannot attain populations equal to the overall mean). Taking out two districts at a time by splitting only a single tract leaves the fewest possible tracts split, which we consider optimal, for the same reasons that number of counties split between districts is optimal.

Our algorithm works as follows. We search for an edge of G such that removal of its two vertices and all edges emanating from them leaves a new graph $G_1 \subset G$ that is connected. We call the deletion of a single vertex from a graph that leaves the graph connected a *paring*. If these two vertices have some special properties, we perform the double paring and then perform the algorithm on G_1, and continue until all districts have equal population. If no such pair of districts exists, we then perform a single paring and ensure that the removed district has population $\bar{p}$ before removing it. Define *tract splitting* to be the process of splitting up a border tract into two disjoint areas and two disjoint populations allocated between two bordering districts.

There always exists an edge on a connected graph G that permits a double paring of G, except for a very specialized set of connected graphs. However, all connected graphs permit a paring, as the next theorem shows.

Theorem 4.1 *All connected graphs permit a paring.*

A proof of this theorem is given in the Appendix B.

We recursively update the districts to get population equality. We iteratively pare the graph G of districts such that each time we pare a district or pair of districts, those districts have populations which equal the population mean. By Theorem 4.1, this process always ends with all districts having equal population. Our algorithm works as follows:

1. If the graph G contains only one district, its population must equal $\bar{p}$. Stop the algorithm here. If not, search across all border tracts of the partition for a tract such that splitting it between two districts makes the population of the two border tracts within 1 of the average $\bar{p}$. If some pair of districts exists which is a double paring of G, then perform this double paring of G. For these two districts, take the tract on their border which, upon being split between the two districts, makes their populations within 1 of the population mean. Split this tract to equalize their populations. If no such pair exists, go to Step 2.

2. Search G for all possible double parings such that the two districts in the double paring have populations which sum to twice the average population. Perform the double paring of G among these double parings which has the property that the two removed districts can have equal populations with the minimal number of tract moves

and one tract splitting between the two. If such a pair exists, perform the double paring and go to Step 1. If no such pair exists, go to Step 3.

3. Search all vertices of G for a paring of G such that a single tract splitting along the border of the district gives the district a population of *barp*, and perform this paring of G. If such a border tract and paring exist, perform the paring and the tract splitting, and go to Step 1. If no such tract splitting and paring exist, go to Step 4.

4. Search all vertices of G for a paring of G such that the removed district D_i borders a district which requires the minimum number of moves and one tract splitting to make the population of D_i equal to $\bar{p}$. Perform these moves, this tract splitting, and this paring, and return to Step 1.

This entire algorithm removes at least one vertex from G at each steps, and the whole algorithm can therefore be performed with at most $n - 1$ tract splittings, where n is the number of districts. The actual number of tract splittings equals $n - d - 1$, where d is the number of double parings performed.

5 Case Study: New York congressional districts

5.1 The data

We began our inquiry by acquiring data from the 2000 census from the New York State Data Center. The downloaded data contained 4907 tracts, but a number of these were tracts have no population. These tracts represented water, inland lakes, or parks. We considered all of these tracts to be the equivalent of water, with the exception of only one of these tracts on Long Island which completely enclosed a populated "island" and was thus considered to be a tract of land with no population. These empty districts are the cause of the "holes" on our maps, particularly around the NYC Metro area.

Trimming these parts from our map left us with 4827 tracts to examine. It is worth noting that the possible number of partitions of these tracts is prohibitively high. Ignoring concerns such as contiguity, nonempty districts, or population equality, the number of allocations of 4827 tracts to 29 districts is approximately

$$\frac{1}{29!} 29^{4827} \approx 1.1 \times 10^{7028}$$

The data were delivered in ESRI shapefile format, which listed tract areas, populations, and unique county identifiers.

5.2 Results

Running the *MSGM* on our initial allocation left us with 29 haggard districts spanning the map from which to refine a solution.

Using this solution as a starting point, we optimized our result using swap moves in particular. Though our algorithmic process of refinement is stochastic, generally more than 90% of the moves in any run involved swaps. This was particularly the case for moves

Table 2: Values after the *MSGM*

Variable	Value
Heuristic Variance Score	-3,147
Largest District	969,511
Smallest District	280,945
County Score	37.48
Compactness Score	2,869

at the very end of a run, where population differences between districts were minute. As a result, swapping provided a way to adjust population smoothly. In addition swap operations, particularly of side-by-side tracts exchanged between districts, provided an effective to "clean up" tattered fringes of districts, increasing their compactness even with vigorous population changes.

Table 3: Values after refinement

Variable	Value
Heuristic Variance Score	-.0277
Largest District	655,760
Smallest District	652,561
County Score	47.44
Compactness Score	2,906

The most difficult part of both steps was defining the optimal values for the scaling factors. It is important to note that it is not the magnitude of the scaling factors that is most crucial, but rather their *relative marginal* magnitudes. Since our algorithm operates on the changes that result from making a single first- or second-order move, selecting positions with the highest score, it is important that the changes in each of the heuristic variables are significant. In particular, a large or small multiple on some factor does not indicate that we wished to treat that variable severely or lightly, but rather that the marginal changes in that variable were relatively small or large.

The Appendix contains several informative tables and maps summarizing our results. Images are produced using the amazing TatukGIS Viewer software.

6 Extension: The 4^{th} Dimension

It is entirely possible that a state's congressional districts could become populationally imbalanced between redistrictings, which usually occur every 10 years. Though current practice is to devise a districting with equal populations per district we suggest that this is suboptimal. One could imagine an initial population allocation that maximizes district population equality not just in the first years but over the course of all 10 years between redistrictings.

For instance, if one district's population is growing 2% a year and another's is shrinking 1% a year then after ten years the two populations will differ by over 33%. With congressional elections occurring every two years it seems arbitary to privilege the population at the year 2000 rather than at the years 2002, 2004, etc. . To improve this disparty we propose starting the growing district with a slightly lower population than that of the shrinking district.

6.1 A stitch in time

For each tract, we can observe certain demographic characteristics, such as race. Based upon population growth estimates from the Census Bureau we can find optimal weighting of populations such that citizens do not have an "equal vote" today, but citizens have the most equal vote over the entire period between each redistrictings.

Let T denote the time between redistrictings; in our case $T = 10$ because the census is taken decenially in the United States. In this section we explore the effect of differential population growth rates by districts on optimal population weights for the districts.

Modern utility theory suggests that individuals favor present utility greater than future utility, and most often, for analytical convenience, according to a constant time discount factor. Let us suppose that the time discount factor for utility of individuals in the United States is given by δ.

We assume that societal utility is maximized by giving citizens an equal voting share in each period. (If this does not actually maximize utility then one could still argue that *ideal* politicians would prefer a scheme that promotes voting share equal.) As we discussed in Section 4.1.1, variance is the best measure for population inequality between districts.

Utility today is weighted greater than utility t units in the future by a factor of $e^{\delta t}$. If we have a partition $\Omega = \{D_1, ..., D_n\}$, with populations $p_1, ..., p_n$, then the population penalty we found for such a partition is a constant multiple of $Var(p_i)$. Let $p_{i,t}$ denote the population of district i at time t. Then the discounted utility of the state at time t is $e^{-\delta t}Var(p_i)$. Suppose that we have forecast data on the population growth rates of different counties during the T-year period. Let the log-growth rate at time t for district i be given by $\eta_{i,t}$. Then the population of district i at time t is given by:

$$p_{i,t} = \exp\left(\int_0^t \eta_{i,s} ds\right) p_{i,0}$$

and total utility of the initial allocation Ω with district populations $\mathbf{p} = (p_1, p_2, ..., p_n)'$ is

$$U_{[0,T]}(\Omega) = -\int_0^T e^{-\delta T}\text{Var}(p_{i,t})dt$$

Expressing the variance in terms of the populations $p_{i,t}$, we get

$$\text{Var}(p_{i,t}) = \frac{1}{n}\sum_{i=1}^{n} p_{i,t}^2 - \frac{1}{n^2}\left(\sum_{i=1}^{n} p_{i,t}\right)^2 = \frac{n-1}{n^2}\sum_{i=1}^{n} p_{i,t}^2 - \frac{1}{n^2}\sum_{i\neq j} p_{i,t}p_{j,t}$$

Dividing out by a constant factor, this gives the total utility as

$$\begin{aligned} U_{[0,T]}(\Omega) &= -(n-1)\sum_{i=1}^{n} p_i^2 \int_0^T \exp\left(2\int_0^t \eta_{i,s}ds - 2\delta t\right)dt \\ &\quad + \sum_{i\neq j} p_i p_j \int_0^T \exp\left(\int_0^t (\eta_{i,s}+\eta_{j,s})ds - 2\delta t\right)dt \end{aligned}$$

This functional form is convenient if we choose to give a specific stochastic process which the logarithmic growth rate may follow. Since our time period is relatively short, we will assume that population growth is simply exponential and thus log-growth rates are constant within our time window, 10 years. So, we set $\eta_{i,s} = \eta_i$ and also define the time-discounted population growth as $\nu_i \equiv \eta_i - \delta$. As long as $\nu_i + \nu_j \neq 0$ then utility simplifies to:

$$U_{[0,T]}(\Omega) = -(n-1)\sum_{i=1}^{n}\left(p_i^2 \int_0^T e^{2(\eta_i-\delta)}dt\right) + \sum_{i\neq j}\left(p_i p_j \int_0^T e^{[(\eta_i-\delta)+(\eta_j-\delta)]t}dt\right)$$

$$= -(n-1)\sum_{i=1}^{n}\left(\frac{e^{2\nu_i T}-1}{2\nu_i}p_i^2\right) + \sum_{i\neq j}\left(\frac{e^{(\nu_i+\nu_j)T}-1}{\nu_i+\nu_j}p_i p_j\right)$$

.

We define the optimal vector of target populations as $\mathbf{p}^* = (p_1^*, ..., p_n^*)^T$ where p_i^* is the optimal population for district i. Under the constraint $\sum_i p_i = P$ (the population of the whole State) we use Lagrange Multipliers to obtain:

$$\lambda = \frac{\partial U_{[0,T]}(\Omega)}{\partial p_i} = -(n-1)\frac{e^{2\nu_i T}-1}{\nu_i}p_i + \sum_{j\neq i} 2\frac{e^{(\nu_i+\nu_j)T}-1}{\nu_i+\nu_j}p_j, 1 \leq i \leq n$$

.

It follows that the vector $\mathbf{p}^*$ satisfies

$$\mathbf{H}\mathbf{p}^* = \lambda\iota$$

,

where $\mathbf{H}$ is the matrix of coefficients

$$\mathbf{H} = \begin{pmatrix} -(n-1)\frac{e^{2\nu_1 T}-1}{\nu_1} & 2\frac{e^{(\nu_1+\nu_2)T}-1}{\nu_1+\nu_2} & \dots & 2\frac{e^{(\nu_1+\nu_n)T}-1}{\nu_1+\nu_n} \\ 2\frac{e^{(\nu_2+\nu_1)T}-1}{\nu_2+\nu_1} & -(n-1)\frac{e^{2\nu_2 T}-1}{\nu_2} & \dots & 2\frac{e^{(\nu_2+\nu_n)T}-1}{\nu_2+\nu_n} \\ \vdots & \vdots & \ddots & \vdots \\ 2\frac{e^{(\nu_n+\nu_1)T}-1}{\nu_1+\nu_n} & 2\frac{e^{(\nu_n+\nu_2)T}-1}{\nu_2+\nu_n} & \dots & -(n-1)\frac{e^{2\nu_n T}-1}{\nu_n} \end{pmatrix}$$

where $\iota = (1, 1, ..., 1)'$ is an $n \times 1$ vector of ones and λ is the Lagrange multiplier.

The expression for $\mathbf{H}$ is analytically convenient as $\mathbf{H}$ is symmetric, and by the Spectral Theorem is orthogonally diagonalizable, enabling a computationally feasible inversion of $\mathbf{H}$ to solve for the optimal populations $\mathbf{p}^*$:

$$\mathbf{p}^* = \lambda\mathbf{H}^{-1}\iota$$

This uniquely determines λ, as the sum of the components of $\mathbf{p}^*$ must be P. We get

$$\lambda = \frac{P}{\iota' \mathbf{H^{-1}} \iota}$$

and this yields the final formula

$$\mathbf{p}^* = \frac{P}{\iota' \mathbf{H^{-1}} \iota} \mathbf{H^{-1}} \iota \tag{3}$$

In the actual implementation, the growth rate η_i is such that if the annual growth rate is g_i, then we have $1 + g_i = e^{\eta_i}$, or

$$g_i = e^{\eta_i} - 1$$

While the estimation of δ is not purely objective, it is reasonable to set the discount rate equal to the discount rate of consumption. In utility-theory analysis, the best measure of the discount rate of consumption is the risk-free interest rate, which is currently best approximated by the overnight lending rate set by the United States Federal Reserve Bank, which is at an annualized $r = 5.25\%$. This implies that if the discount rate is δ, then δ is given by $e^{\delta} = 1 + r$, or

$$\delta = \log(1 + r) \approx 5.1168\%$$

We use this rough approximation in the following section.

6.2 Implementation of the extension

We are using data from the 2000 census, so to estimate the population growth rates in the 2000-2010 redistricting period, we use realized *county* population growth rates during the 2000-2003 period.

The output of our model gives allocations based on equal population and we estimate the population growth rates of the *districts* by assuming uniform population growth rates within each county. It is easy to calculate how much each district is made up of various counties and we use these proportions as weights to approximate the *district population growth rate* as a weighted average of *county population growth rates.*[1]

Based on the optimal population vector $\mathbf{p}^*$ found via Equation (3) we can rerun *POM* with the populations goal of $\mathbf{p}^*$. This procedure can be iterated as: run the *POM*, find the growth rates of each district produced, calculate the optimizations of initial populations based on the above theory, and feed the results back into *POM*. We settle on a final districting plan when the solution converges within some reasonable bound.

Figure 4 shows one iteration of this process. The initial result from *POM* is p_i and district growth rates are found using our Census data about county growth rates. The final column shows the optimal initial population that from Equation (3) that will maximize societal voting equality over the entire period between redistrictings. One can easily see

[1]We are assuming that district growth rates remains constant over time which is inconsistent with our previous assumption that the county growth rates are constant. This is a small, simplifying assumption and the interested reader may make these assumptions consistent by explicitly calculating district growth rates over time in terms of the county growth rates and initial population distribution of counties in districts. The theory above, using stochastic logarithmic growth rates, is designed to accomodate such generalizations.

that districts with higher projected growth rates (η_i) are assigned lower optimal starting populations.

The results make intuitive sense: faster growing districts are initially under-allocated and slower growing districts are over-allocated in terms of starting population. There is a significant effect of taking into account population changes over time. The difference between the smallest and largest optimal district populations is 69,133, which is 10.6% of the total average district population. This implies that, with a reasonable level of certainty about future population growth rates, it may be beneficial for legislators to take future population growth into account when redistricting.

Table 4: District Population Growth Rates

p_i	Est. 2003 population	η_i	Optimal initial pop.
655,067	681,997	2.01%	613,786
654,373	678,814	1.83%	618,869
655,760	678,245	1.69%	622,818
654,715	673,058	1.38%	631,544
654,449	668,395	1.05%	640,802
654,140	667,976	1.05%	640,802
655,184	668,555	1.01%	641,922
653,486	666,411	0.98%	642,761
653,636	665,372	0.89%	645,278
654,702	665,452	0.81%	647,513
654,164	664,307	0.77%	648,630
653,902	663,066	0.70%	650,582
653,884	662,672	0.67%	651,418
652,561	659,850	0.56%	654,482
655,040	660,798	0.44%	657,818
654,383	659,926	0.42%	658,374
653,655	656,265	0.20%	664,474
655,311	657,351	0.16%	665,581
653,676	655,585	0.15%	665,857
653,792	655,701	0.15%	665,857
654,568	656,471	0.15%	665,857
654,739	656,443	0.13%	666,411
654,745	655,765	0.08%	667,793
654,041	654,476	0.03%	669,173
654,834	654,665	-0.01%	670,277
654,381	654,019	- 0.03%	670,829
654,242	653,298	-0.07%	671,932
654,395	648,670	-0.44%	682,097
654,632	648,514	-0.47%	682,919

In the above, p_i is the value that our model returns for the population of the 29 districts. The estimated 2003 populations are calculated for each district based on county growth rates. One can easily see that districts with higher projected growth rates (η_i) are assigned lower optimal starting populations.

7 Analysis of the Models

7.1 Solving the Problem

By combining the *Multi-seeded Growth Model* with the *Partition Optimization Model* we effectively devised a strategy for creating fair and geometrically compact congressional districts. The districts conform to several well accepted measures of district goodness: population equality, contiguity, preservation of county boundaries, and compactness of shape.

The districts produced by our models are both simple and fair. Geometric *simplicity* is measured by compactness, as determined by how close the members of a districts live realtive to each other. Additionally, our method penalizes splitting counties between several districts so that nearby citizens, who have simliar concerns, will be represented by the same congressperson. The *fairness* of our methodology is evident in its perfect indifference to partisan politics, incumbent protection, and race/ethnicity.

We apply our models to create a congressional district partition of New York State based on 2000 US Census Bureau data. The results in Figures 6, 8, and 10 clearly demonstrate a partitioning into contiguous, compact, and reasonable districts. Furthermore, the simulations that produced these visually pleasing results also achieved extremely high degrees of population equality and county preservation.

7.2 Strengths of Model

The model successfully generates district partitions that simultaneously excel against the standard metrics of county integrity, compactness, and population equality. Unlike other models in the literature, we provide an algorithm for reducing population differences to at most 1 by breaking up a minimal number of tracts.

We also find that in order to equalize population of the districts as much as possible, any knowledge about future district growth rates yields highly unequal initial district populations, contrary to one of the fundamental assumptions of all existing algorithms in the literature.

The model runs independently of the distribution of population, and works well both in low- and high- density locales, and with regular and oddly shaped census tracts. This is evidenced by the successful districtings that our model produces in rural, small city, and large metropolitan areas. (See the Figures 5 through 10.)

The algorithm runs efficiently enough that it can generate districts for large States, such as New York (population: 18,976,457), in a run time of less than an hour.

7.3 Weaknesses of Model

The model assumes contiguity of the entire State so in cases where contiguity cannot be forced, such as Hawaii or Michigan, we must change the algorithm slightly. One solution could be to divide the State into several regions and run our model separately on each region, allocating the porportionally correct number of representatives to each region based on population.

A second limitation is that the model appears to tend toward creating districts that are either very low- or high-density, instead of splitting smaller population centers into a number of districts. As political affiliation and race are likely correlated with population density, the algorithm may inadvertently generate districts which separate various demographic groups into separate districts, which could be viewed as gerrymandering. Yet, another camp would argue that it is appropriate to divide urban, suburban, and rural areas into separate districts since their residents have different concerns.

7.4 Future Investigations

A problem with any computer-based solution to the redistricting problem is that the methodology used in the redistricting algorithm may indirectly lead to some form of gerrymandering. Because the program is not deterministic and can be evaulated many times, the entity running the program should not be able to arbitrarily choose a result as this could be characterized as gerrymandering. (We tie our hands by choosing the highest scoring result based on our goodness metric but a future modeller with an ulterior motive could be less objective.)

To solve this we should test our simulations and throw out any results that, by random chance, display the qualities of partisan or racial/ethnic gerrymandering. This could be done relatively easily by merging tract level data with data political and racial characteristics.

This model sought to create a baseline alternative to the political misuse of congressional districting, but it could be expanded to a loftier goal. For instance, we assume that race/ethnicity should play no role in creating districts but it is conceivable that citizens are better off when minority groups control a few districts so that these groups are guaranteed at least a few representatives. If every district is a perfect cross-section of the State's demographics then minority groups will have *ex ante* equal political power but not *ex post*. More work needs to be done to understand the legal, philosophical, and mathematical underpinnings of districting in a representative democracy.

An open letter concerning congressional districting

TO: Sheldon Silver, Assembly Speaker, New York State Assembly
CC: Robert D. Lenhard, Chairman, Federal Election Comission
CC: Rex Smith, Editor, Albany Time Union
FROM: MCM Team # 1421
DATE: February 12, 2007

The negative consequences of Gerrymandering are well accepted: voters become apathetic, minority groups are sequestered to a few districts, and the political process moves farther and farther from the electorate's best interests. We present to the you, the Assemblymen and Assemblywomen of New York, a new method to create fair districts with simple shapes that citizens will appreciate and embrace.

We have devised a set of rules that a computer can implement to create districts that are:

1. Contiguous - there are no breaks in the district lines
2. Equally sized in population
3. Conscious of county boundaries - especially in upstate New York congressional districts will avoid splitting county lines
4. Compact - districts are not winding, long and skinny, or oddly shaped

Our scheme produces fair districts in that choices are made without prejudice or favor to residents of particular racial, ethnic, or socioeconomic groups. At the same time, by producing districts that break up the fewest possible tracts, we ensure that voters with roughly similar characteristics and geographical location will be represented by the same congressperson. This has the effect of encouraging civic involvement by residents, aligning representatives' interests with those of their consituents, and fostering a healthier democracy.

By implementing our redistricting method, the Empire State can be a pioneer in guaranteeing the rights of its citizens. Since the 19^{th} Century, Elbridge Gerry's lizard has grown into a terrible, twisting serpent, eating away at our Democracy.

It is time to put Gerrymanders on a healthier diet.

References

[1] Barkan, J. D., P. J. Densham, and G. Rushton (2006). Space Matters: Designing Better Electoral Systems for Emerging Democracies. *American Journal of Political Science, 50* (4), 926-939.

[2] Bong, C. and Y. Wang (2006). A multi-objective hybrid metaheuristic for zone definition procedure. *Int. J. Services Operations and Informatics (1)* (1/2), 146-164.

[3] Caliper Corporation. "About Census Summary Levels." Available at http://www.caliper.com/Maptitude/Census2000Data/SummaryLevels.htm.

[4] Cirincione, C., T. A. Darling, and T. G. O'Rourke (2000). Assessing South Carolina's 1990s Congressional Redistricting. *Political Geography, 19*, 189-211.

[5] Garfinkel, R. S. and G. L. Nemhauser (1970). Optimal political districting by implicit enumeration techniques. *Management Science, 16* (4), B495-B508.

[6] Hunt v. Cromartie, 526 U. S. 541 (1999).

[7] Karcher v. Daggett, 462 U.S. 725 (1983).

[8] Kaiser, H. (1966). An objective method for establishing legislative districts. *Midwest Journal of Political Science, 10.*

[9] League of United Latin American Citizens v. Perry, 548 U.S. ___ (2006).

[10] Luttinger, J. M. (1973). Generalized Isoperimetric Inequalities. *Proceedings of the National Academy of Sciences of the United States of America, 70*, 1005-1006.

[11] Macmillan, W. (2001). Redistricting in a GIS environment: An optimisation algorithm using switching-points.

[12] Macmillan, W. and T. Pierce (1994). Optimization modeling in a GIS framework: the problem of political districting. In S. Fotheringham and P. Rogerson, *Spatial Analysis and GIS*. Bristol: Taylor and Francis. *Journal of Geographical Systems, 3*, 167-180.

[13] Mehrotra, A., E. L. Johnson and G. L. Nemhauser (1998). An Optimization Based Heuristic for Political Districting. *Management Science, 44* (8), 1100-1114.

[14] Miller v. Johnson, 515 U. S. 900 (1995).

[15] NationalAtlas.gov Maps of US Congressional Districts.

[16] Nagel, S. (1965). Simplified bipartisan computer redistricting. *Stanford Law Review, 17*, 863-899.

[17] ew York State Data Center. "2000 Census Data." Available at http://www.empire.state.ny.us/nysdc/.

[18] Shaw v. Reno , 509 U. S. 630 (1993).

[19] SC State Conference of Branches, Etc. v. Riley (1982). 533 F. Supp. 1178 (DSC). Affirmed 459 US 1025.

[20] S Census Bureau. “New York county data 2000-2003.” Available at http://www.epodunk.com/top10/countyPop/coPop33.html.

[21] Weaver, J. B. and S. W. Hess (1963). A procedure for nonpartisan districting: development of computer techniques. *The Yale Law Journal, 73* (1), 287-308.

[22] ahoo Finance. Market data on US Treasury bond rates. Available at http://finance.yahoo.com/bonds/composite_ bond_ rates .

[23] Young, H. P. (1988). Measuring the Compactness of Legislative Districts. *Legislative Studies Quarterly.* **XIII** 105-115.

A Tables and Maps

Table 5: Final partition of counties after the *POM*. f is the fraction of the county allocated to the largest district in that county, while d represents the number of the districts with tracts in that county.

County Name	f	d	County Name	f	d
Albany	0.84	2	Niagara	1	1
Allegeny	1	1	Oneida	1	1
Bronx	0.74	4	Onondaga	0.94	2
Broome	0.71	2	Ontario	1	1
Cattaraugus	0.53	3	Orange	0.85	2
Cayuga	0.94	2	Orleans	1	1
Chautauqua	1	1	Oswego	0.92	2
Chemung	0.52	3	Otsego	0.59	2
Chenango	0.83	3	Putnam	1	1
Clinton	1	1	Queens	1	1
Columbia	0.9	2	Rensselaer	0.87	3
Cortland	1	1	Richmond	1	2
Delaware	0.56	2	Rockland	1	1
Dutchess	1	1	Saratoga	1	1
Erie	1	1	Schenectady	0.83	2
Essex	1	1	Schoharie	0.93	2
Franklin	1	1	Schuyler	0.64	6
Fulton	0.55	5	Seneca	1	1
Genessee	0.43	9	St. Lawrence	1	1
Greene	0.33	13	Steuben	1	1
Hamilton	0.53	7	Suffolk	0.81	2
Herkimer	1	1	Sullivan	1	1
Jefferson	0.42	9	Tioga	0.86	2
Kings	0.27	13	Tompkins	1	1
Lewis	0.88	3	Ulster	0.92	2
Livingston	1	1	Warren	0.71	2
Madison	0.75	2	Washington	0.6	4
Monroe	1	1	Wayne	0.51	4
Montgomery	1	1	Westchester	0.73	4
Nassau	1	2	Wyoming	1	1
New York	0.97	2	Yates	1	1

Averages: $f = .85$, $d = 2.55$

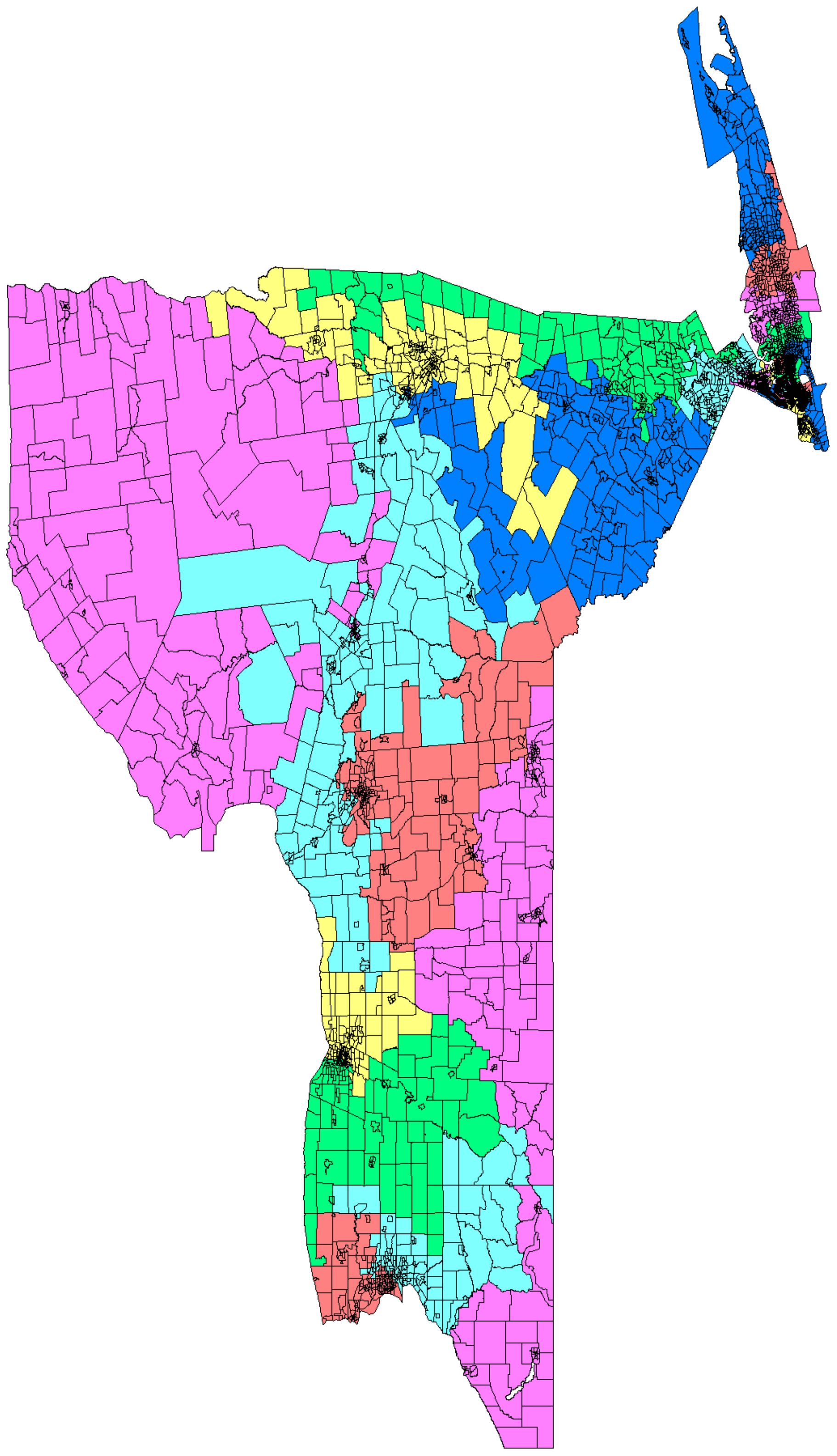


Figure 5: New York congressional districts from the *MSGM* (initialized districts)

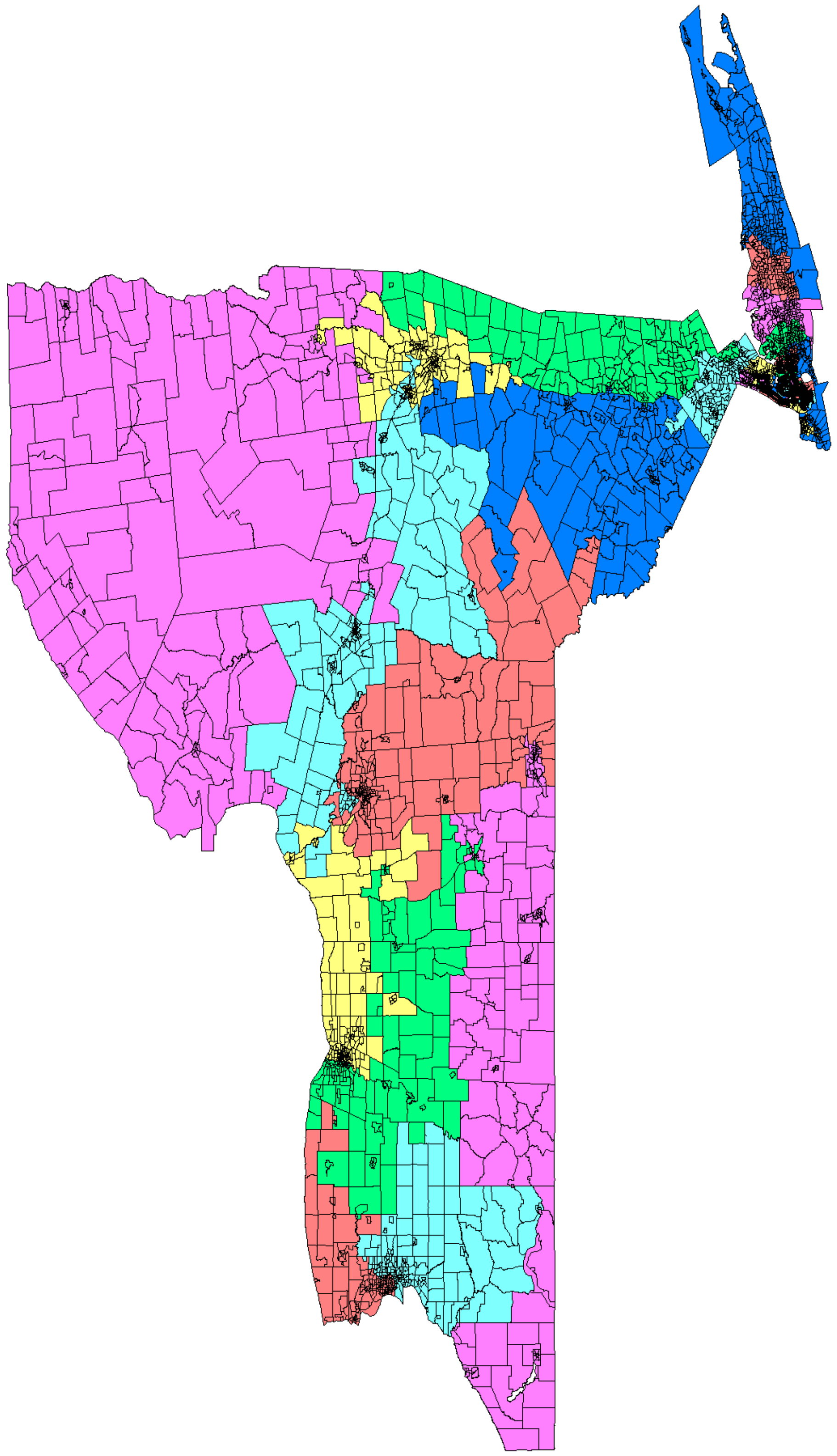


Figure 6: New York congressional districts from the *POM* (final optimization)

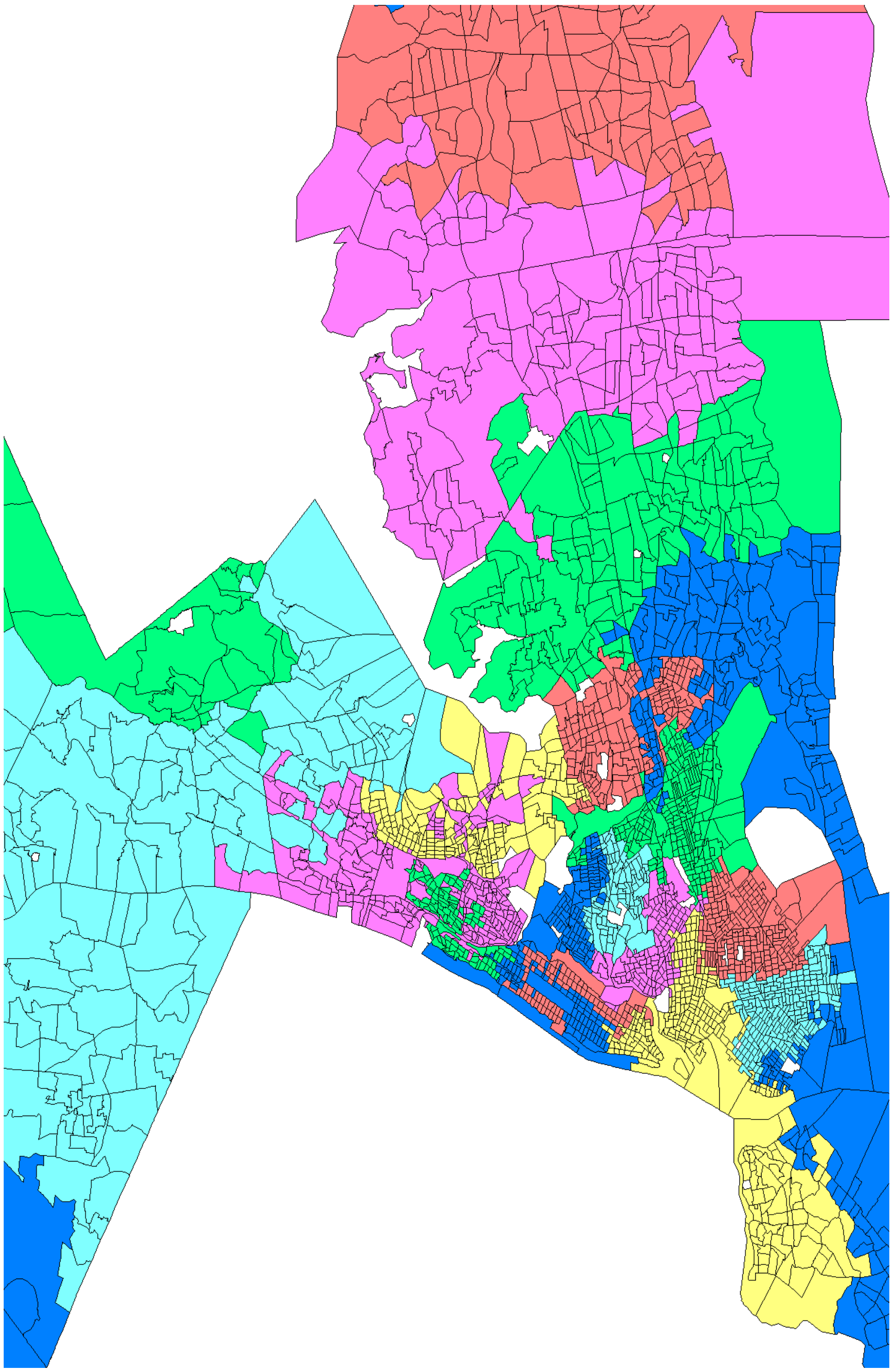


Figure 7: NYC metro-area *MSGM* (initialized districts)

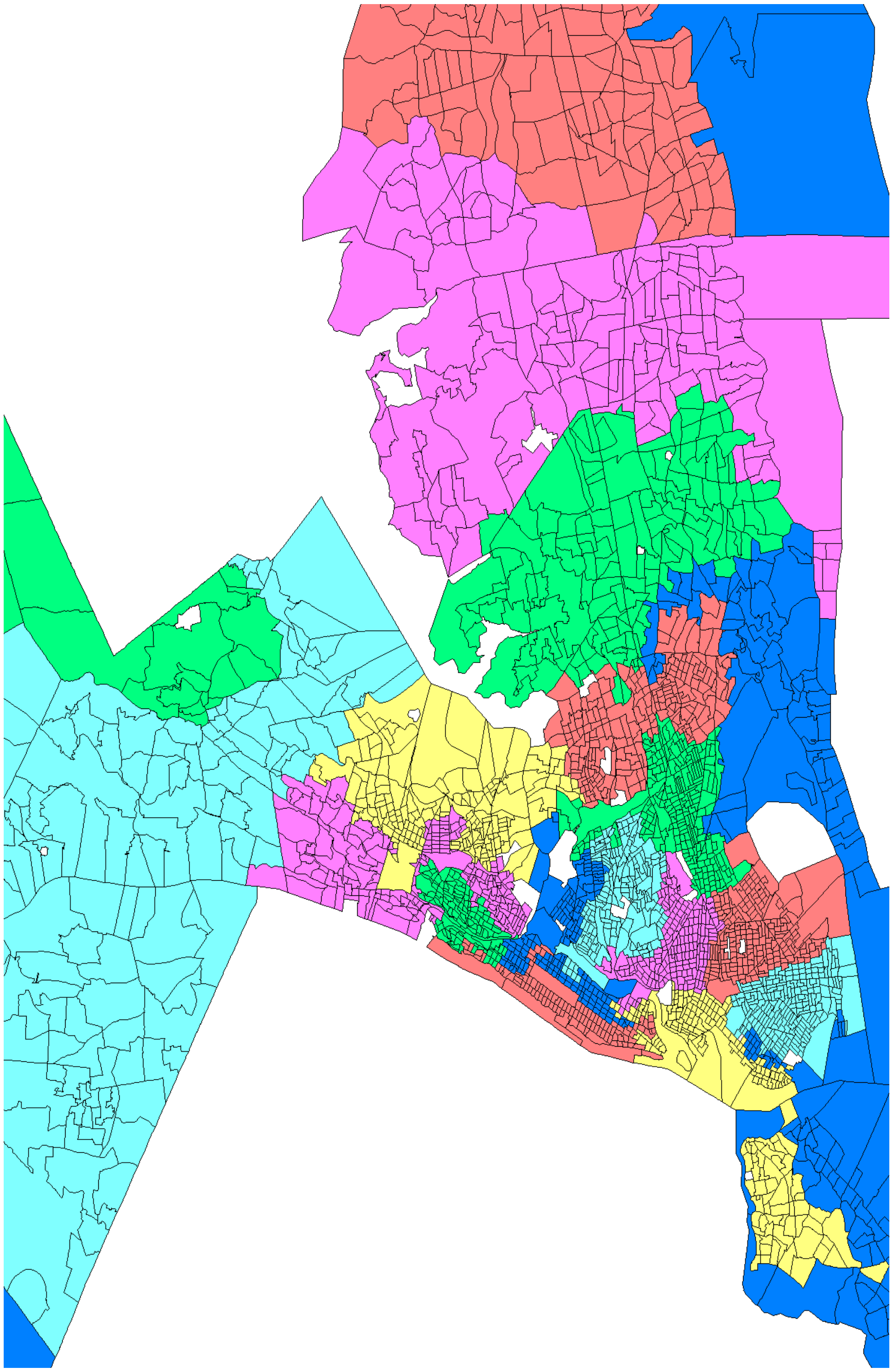


Figure 8: NYC metro-area *POM* (final optimization)

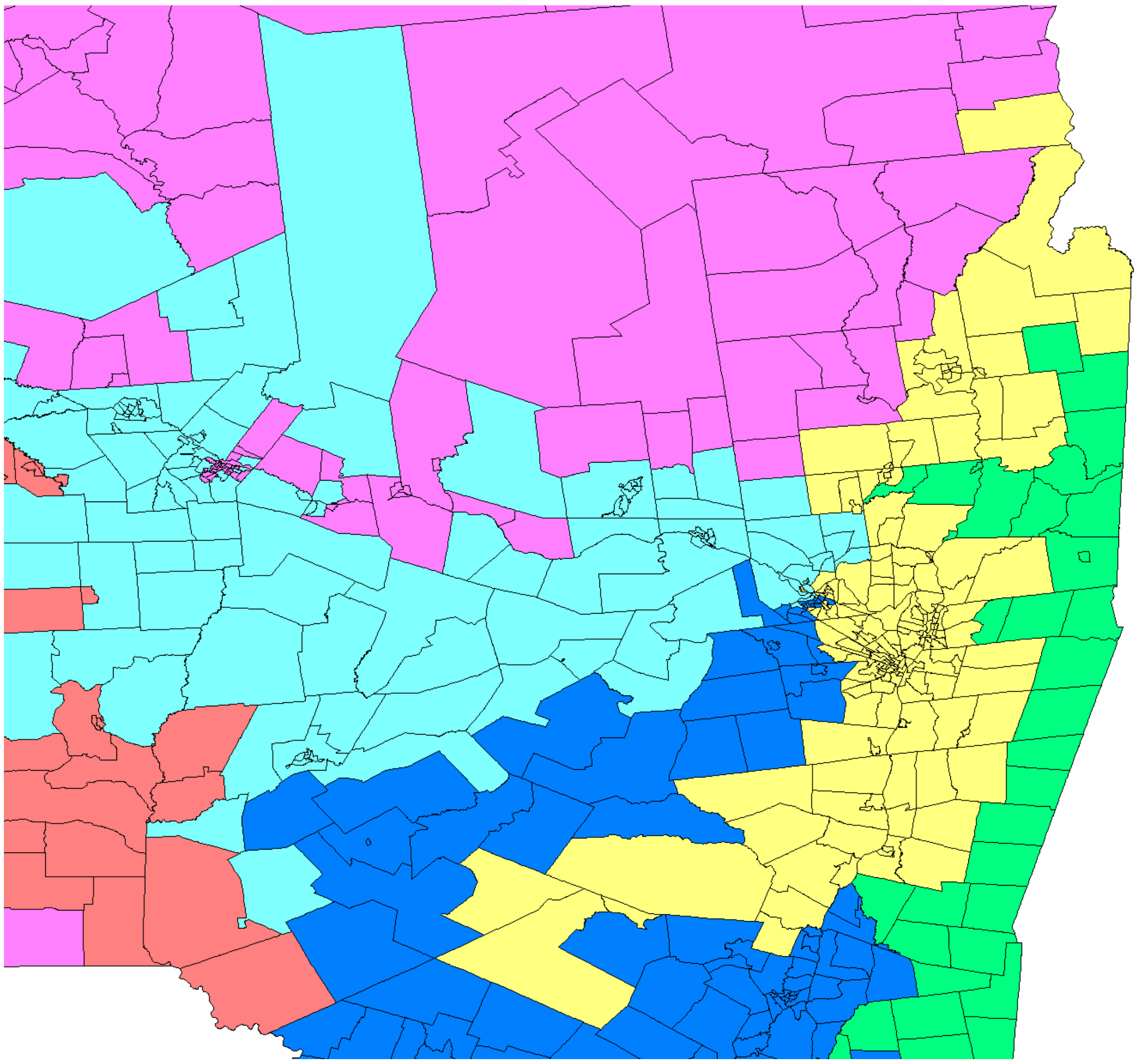


Figure 9: Close-up of the Albany area *MSGM* (initialized districts)

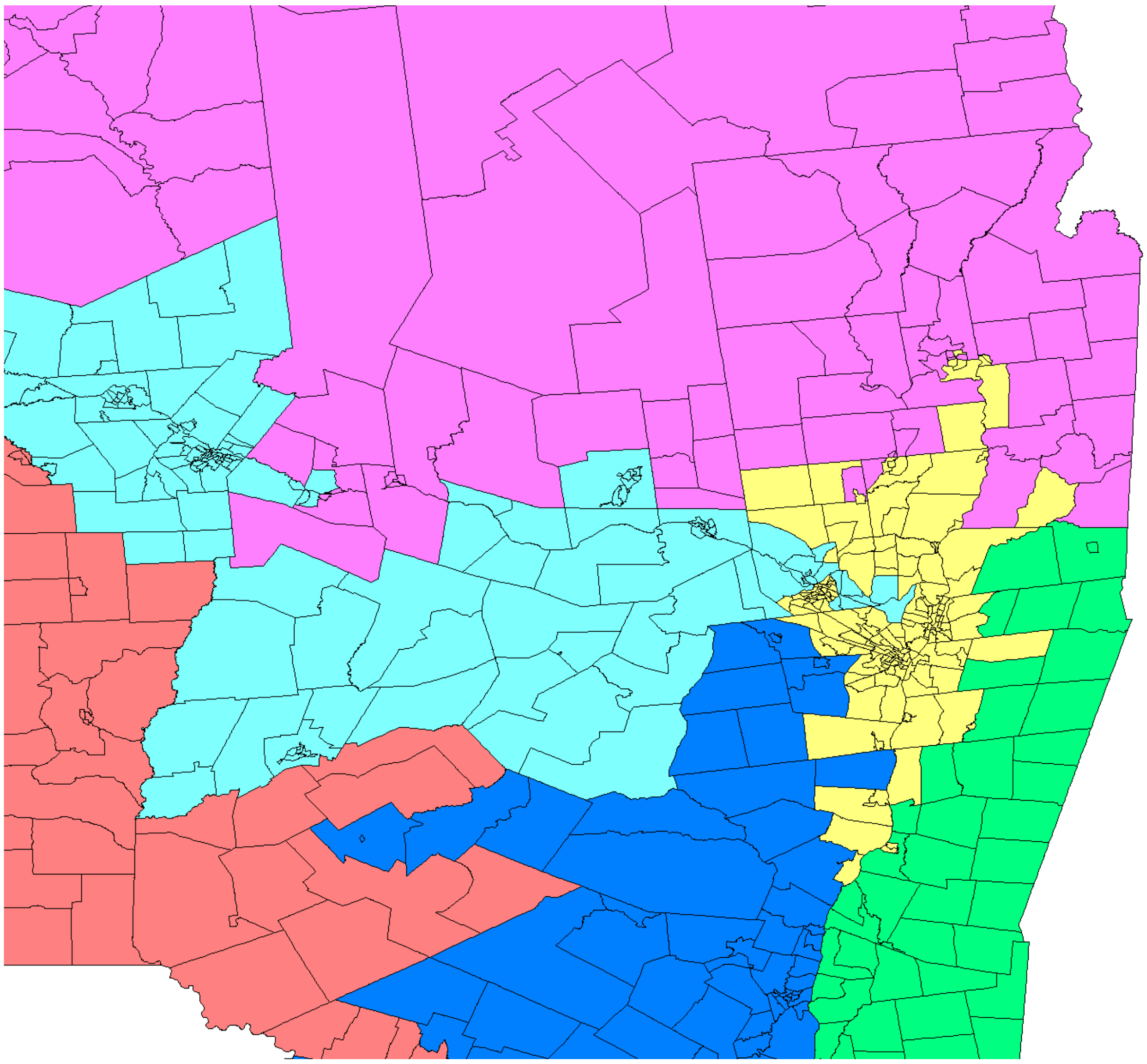


Figure 10: Close-up of the Albany area *POM* (final optimization)

B Proof of Theorem 4.1

Theorem B.1 *All connected graphs permit a paring.*

Proof We prove that any connected graph G permits a paring, by induction on the number of vertices y. We prove a stronger statement, namely that for any connected graph G with at least two vertices, there exist at least two parings. The claim clearly holds for $y = 2$.

Suppose the claim holds for $y = k$, where $k \geq 2$. Then for $y = k + 1$, suppose the claim does not hold. Then as $y \geq 3$, take any vertex v of G such that removal of v leaves G unconnected, and consider two disjoint subgraphs G_1, G_2 into which G is divided upon removal of this vertex. By the induction hypothesis, there exist vertices v_1, v_2 of G_1 such that its removal leaves G_1 connected.

I claim that removal of one of v_1, v_2 from the original graph G leaves G connected. To see this, note that neither v_1 nor v_2 is adjacent to any vertex in G_2, as G_1, G_2 have no common edges. If both v_1, v_2 are adjacent to v, then removal of v_1 leaves G connected. This is because if we let $G' = G - \{v_1\}$ and $G'_1 = G_1 - \{v_1\}$, then G' consists of $G'_1 \cup \{v\}$ and G_2, which are both connected and connected to each other, as v is necessarily connected to G_2.

This means that $G - \{v_1\}$ is connected. If one of v_1, v_2 is not adjacent to v_1, WLOG assume it is v_1. Then removing v_1 from G leaves the graph connected, as $G'_1 \cup \{v\}$ is connected, as is G_2, and they are connected to each other. Some such vertex which admits a paring also exists in G_2, yielding two vertices which permit a paring. This proves the result by induction. ∎

C Computer codes

```
// Tract.h - header file for a Tract
// a Tract has an area, a perimeter, a population, an ID, and a county.
// and an OGRGeometry...

#ifndef TRACT_H
#define TRACT_H

#include <iostream>
#include <vector>
#include <string>
#include <cmath>

class County;
class District;

using namespace std;

class Tract {
    protected:
        double _area;
        double _perim;
        int _population;
        string _id;
        int _county;
        int _index;
        OGRGeometry *_geo;
        OGRPoint *_centroid;
        vector<Tract *> neighbors;
        District *_mydist;
        County *_mycounty;
        map<Tract *,double> shared;

    public:
        Tract(){ }
        Tract(OGRFeature *me, int index){
            _area = me->GetFieldAsDouble(me->GetFieldIndex("AREA"));
            _population =
                me->GetFieldAsInteger(me->GetFieldIndex("TOTALPOP"));
            _id = me->GetFieldAsString(me->GetFieldIndex("ID"));
            string::size_type  notwhite = _id.find_first_not_of(" \t\n");
            _id.erase(0,notwhite);

            // trim trailing whitespace
            notwhite = _id.find_last_not_of(" \t\n");
            _id.erase(notwhite+1);
            _county =
                me->GetFieldAsInteger(me->GetFieldIndex("COUNTYFP"));
            _geo = me->StealGeometry();
            _centroid = new OGRPoint();
            ((OGRPolygon *)_geo)->Centroid(_centroid);
            _index = index;
            _perim = (((OGRPolygon *)_geo)->getExteriorRing())->get_Length();
        }

        // Setters
        void addPerim(Tract *t,double d){
                shared[t] = d;
        }

        void setCounty(County *c){
            _mycounty = c;
        }

        void setN(const vector <Tract *> &n){
            int i;
            for(i=0; i < n.size(); i++){
                if((n[i]->getPop() == 0) && (n[i]->getID() != "1491835"))
                    continue;
                neighbors.push_back(n[i]);
            }
        }

        void setDistrict(District *d){
            _mydist = d;
        }

        // Getters
        double getShared(Tract *t){ return shared[t]; }
        County* getMyCounty(){ return _mycounty; }
        int getIndex(){ return _index; }
        vector<Tract *> getN(){ return neighbors; }
        District *getDistrict(){ return _mydist; }
        double getArea(){ return _area;}
        double getPerim(){ return _perim;}
        int getPop(){ return _population;}
        string getID(){ return getId();}
        string getId(){ return _id;}
        int getCounty(){ return _county;}
        OGRPoint* getCentroid(){ return _centroid;}
        OGRGeometry *getGeo(){
            return _geo;
        }

        // Neat stuff I can do with Tracts
        double bcMetric(Tract *t){
            return distC(t)/min(getArea(),(t->getArea()));
        }

        double getPopDen(){
            return getPop()/getArea();
        }

        bool bordersp(Tract *t){
            OGRGeometry *g = t->getGeo();
            return _geo->Touches(g);
        }

        double distBetweenTracts(Tract *t){
            OGRGeometry *g = t->getGeo();
            return _geo->Distance(g);
        }

        double dist(OGRPoint *oc){
            double xdiff = _centroid->getX() - oc->getX();
            double ydiff = _centroid->getY() - oc->getY();
            return sqrt(xdiff*xdiff + ydiff*ydiff);
        }

        double distC(Tract *t){
            OGRPoint *oc = t->getCentroid();
            double xdiff = _centroid->getX() - oc->getX();
            double ydiff = _centroid->getY() - oc->getY();
            return sqrt(xdiff*xdiff + ydiff*ydiff);
        }

        bool onPerimeter(){
            // returns true iff exists a neighboring tract with a
            // different district assignment
            vector<Tract *>::iterator iter;
            for(iter = neighbors.begin(); iter != neighbors.end(); iter++){
                if((*iter)->getDistrict() != _mydist){
                    return true;
                }
            }
            return false;
        }

        vector <District *> getNColors(){
            // returns list of districts touching this one...
            int i;
            vector<Tract *> n = getN();
            map<District *,bool> seenit;
            vector<District *> retval;

            for(i=0; i < n.size(); i++){
                if((n[i]->getDistrict() != getDistrict()) &&
                        !seenit[n[i]->getDistrict()]){
                    retval.push_back(n[i]->getDistrict());
                    seenit[n[i]->getDistrict()] = true;
                }
            }
            return retval;
        }

};

#endif
```

```

// Fnode.h - defines a Fronteir node structure, consisting of a Tract
// and the District to change that Tract to.

#include <iostream>
#include "Tract.h"
#include "District.h"

class Fnode {
        private:
                Tract *_t;
                District *_d;
                double _score;

        public:
                Fnode() { }
                Fnode(Tract *t, District *d){
                        _t = t;
                        _d = d;
                }

                void setScore(double score){
                        _score = score;
                }

                double getScore(){
                        return _score;
                }

                Tract *getTract(){
                        return _t;
                }

                District *getDistrict(){
                        return _d;
                }
};
```

```
// County.h - header file for a County
// a County consists of a list of pointers to tracts.

#ifndef COUNTY_H
#define COUNTY_H

```

```
#include <iostream>
#include <vector>
#include <map>
#include "Tract.h"

class District;

using namespace std;
// NDIST?
extern District* BLANKDIST;

class County {
    protected:
        vector<Tract *> myTracts;
        double area;
        int population;

    public:
        County(){
            population = 0;
            area = 0;
        }
        void addToCounty(Tract *t){
            myTracts.push_back(t);
            area += t->getArea();
            population += t->getPop();
        }

        void printCounty(){
            //map<District *,int> p;
            map<District *,double> a;

            map<int,string> cnames;
            cnames[3] = "Allegeny";
            cnames[13] = "Chautauqua";
            cnames[9] = "Cattaraugus";
            cnames[29] = "Erie";
            cnames[63] = "Niagara";
            cnames[73] = "Orleans";
            cnames[37] = "Genesee";
            cnames[121] = "Wyoming";
            cnames[55] = "Monroe";
            cnames[51] = "Livingston";
            cnames[117] = "Wayne";
            cnames[101] = "Steuben";
            cnames[69] = "Ontario";
            cnames[123] = "Yates";
            cnames[11] = "Cayuga";
            cnames[97] = "Schuyler";
            cnames[99] = "Seneca";
            cnames[15] = "Chemung";
            cnames[33] = "Franklin";
            cnames[109] = "Tompkins";
            cnames[107] = "Tioga";
            cnames[23] = "Cortland";
            cnames[75] = "Oswego";
            cnames[45] = "Jefferson";
            cnames[89] = "St. Lawrence";
            cnames[49] = "Lewis";
            cnames[67] = "Onondaga";
            cnames[7] = "Broome";
            cnames[17] = "Chenango";
            cnames[43] = "Herkimer";
            cnames[41] = "Hamilton";
            cnames[31] = "Essex";
            cnames[113] = "Warren";
            cnames[19] = "Clinton";
            cnames[115] = "Washington";
            cnames[83] = "Rensselaer";
            cnames[21] = "Columbia";
            cnames[27] = "Dutchess";
            cnames[91] = "Saratoga";
            cnames[35] = "Fulton";
            cnames[93] = "Schenectady";
            cnames[57] = "Montgomery";
            cnames[25] = "Delaware";
            cnames[77] = "Otsego";
            cnames[65] = "Oneida";
            cnames[53] = "Madison";
            cnames[21] = "Columbia";
            cnames[27] = "Dutchess";
            cnames[79] = "Putnam";
            cnames[119] = "Westchester";
            cnames[105] = "Sullivan";
            cnames[71] = "Orange";
            cnames[111] = "Ulster";
            cnames[39] = "Greene";
            cnames[95] = "Schoharie";
            cnames[1] = "Albany";
            cnames[87] = "Rockland";
            cnames[103] = "Suffolk";
            cnames[59] = "Nassau";
            cnames[81] = "Queens";
            cnames[85] = "Richmond";
            cnames[47] = "Kings";
            cnames[5] = "Bronx";
            cnames[61] = "New York";

            int i;
            for(i=0; i < myTracts.size(); i++){
                //p[myTracts[i]->getDistrict()] += myTracts[i]->getPop();
                a[myTracts[i]->getDistrict()] += myTracts[i]->getArea();
            }

            double x;
            double largest = -1;
            //map<District *,int>::iterator piter;
            map<District *,double>::iterator aiter;
            cout << cnames[myTracts.front()->getCounty()] << " ";
            for(aiter = a.begin(); aiter != a.end(); aiter++){
                x = (double)(aiter->second)/(double)getArea();
                if(x > largest){
                        largest = x;
                }
            }
            cout << largest << " " << a.size() << endl;
        }

        vector<Tract *> getTractList(){
            return myTracts;
        }

        int getPop(){
            return population;
        }

        double getArea(){
            return area;
        }

        double getValue(){
            double scale = 1e7;

            map<District *,int> p;
            //map<District *,double> a;

```

```
            double a = 1.0 * scale;

            int i;
            for(i=0; i < myTracts.size(); i++){
                if(myTracts[i]->getDistrict() != BLANKDIST){
                    p[myTracts[i]->getDistrict()] += myTracts[i]->getPop();
                }
                //a[myTracts[i]] += myTracts[i]->getArea();
            }

            double returnval=0;
            double x;
            map<District *,int>::iterator piter;
            //map<District *,double>::iterator aiter;

            for(piter = p.begin(); piter != p.end(); piter++){
                x = (double)(piter->second)/(double)population;
                returnval += a*x*x;
            }

            return returnval;
        }

};

#endif
```

```

// District.h - header file for a District
// a District consists of a list of tracts, area, perimeter, and
// population.

#ifndef DISTRICT_H
#define DISTRICT_H

#include <iostream>
#include <list>
#include <map>
#include <vector>
#include "Tract.h"
#include "County.h"
#include <sstream>


using namespace std;
extern District *BLANKDIST;
extern const double AVGPEOPLE;
extern bool comp_func(Tract *lhs, Tract *rhs);
extern bool eq_func(Tract *lhs, Tract *rhs);

class District {
    protected:
        list<Tract *> myTracts;
        double _area;
        double _perimeter;
        int _population;
        int _numtracts;

    public:
        District() {
            _area = 0;
            _population = 0;
            _numtracts = 0;
        }

        void removeFromDistrict(Tract *t){
            myTracts.remove(t);
            _numtracts--;
            _area =  _area - t->getArea();
            _population = _population - t->getPop();
        }

        void addToDistrict(Tract *t){
            if((t->getPop() == 0) && (t->getID() != "1491835"))
                return;
            myTracts.push_front(t);
            _numtracts++;
            _area += t->getArea();
            //_perimeter += t->getPerimeter();
            // would need to do pairwise elimination on borders...
            _population += t->getPop();
        }

        double getArea(){
            return _area;
        }

        /*
           double getPerimeter(){
           return _perimeter;
           }*/

        double getIsoPerim(){
            double scale = .001;
            OGRGeometry *uni;
            list<Tract *>::iterator liter;
            list<Tract *> l = getPerimeter();
            vector<Tract *> n;
            double p=0;
            int i;
            double count;
            //uni = ((myTracts.front())->getGeo())->clone();

            for(liter = l.begin(); liter != l.end(); liter++){
                count = 0;
                n = (*liter)->getN();
                for(i=0; i < n.size(); i++){
                    if(n[i]->getDistrict() != this ){
                        count++;
                    }
                }
                p += ((*liter)->getPerim())*(count/(double)n.size());
            }
            /*
            for(liter = myTracts.begin(); liter != myTracts.end();
                    liter++){
                n = (*liter)->getN();
                p = p + (*liter)->getPerim();
                for(i=0; i < n.size(); i++){
                    if(n[i]->getDistrict() == this){
                        p += n[i]->getPerim() - (*liter)->getShared(n[i]);
                    }
                }
            }*/

            //double a = ((OGRPolygon *)uni)->get_Area();
            double a = getArea();
            //OGRLinearRing *perim = ((OGRPolygon*)uni)->getExteriorRing();
            //double p = perim->get_Length();
            //cout << "Area " << a << " Perimeter " << p << endl;
            //delete uni;

            return scale*a/(p*p);
        }


        int getPop(){
            return _population;
        }

        int getNumTracts(){
            return _numtracts;
        }

        list<Tract *> getTractList(){
            return myTracts;
        }

        double score(){
            return newcountyScore() + compactScore() + varScore() + countyScore();
        }

        double newcountyScore(){
            double M = 0.;
            list<Tract *>::iterator liter;
            map<County *,double> pz;
            County *c;
            double frac;
            map<County *,double>::iterator miter;
            double retval=0;

            for(liter = myTracts.begin(); liter != myTracts.end();
                    liter++){
                c = (*liter)->getMyCounty();
                pz[c] += (*liter)->getArea();
            }

            return M*pz.size();
        }

        double countyScore(){
            double M = 1.;
            list<Tract *>::iterator liter;
            map<County *,double> pz;
            map<int,string> cnames;
            County *c;
            double frac;
            map<County *,double>::iterator miter;
            double retval=0;
// Initialize County Names
            cnames[3] = "Allegeny";
            cnames[13] = "Chautauqua";
            cnames[9] = "Cattaraugus";
            cnames[29] = "Erie";
            cnames[63] = "Niagara";
            cnames[73] = "Orleans";
            cnames[37] = "Genesee";
            cnames[121] = "Wyoming";
            cnames[55] = "Monroe";
            cnames[51] = "Livingston";
            cnames[117] = "Wayne";
            cnames[101] = "Steuben";
            cnames[69] = "Ontario";
            cnames[123] = "Yates";
            cnames[11] = "Cayuga";
            cnames[97] = "Schuyler";
            cnames[99] = "Seneca";
            cnames[15] = "Chemung";
            cnames[33] = "Franklin";
            cnames[109] = "Tompkins";
            cnames[107] = "Tioga";
            cnames[23] = "Cortland";
            cnames[75] = "Oswego";
            cnames[45] = "Jefferson";
            cnames[89] = "St. Lawrence";
            cnames[49] = "Lewis";
            cnames[67] = "Onondaga";
            cnames[7] = "Broome";
            cnames[17] = "Chenango";
            cnames[43] = "Herkimer";
            cnames[41] = "Hamilton";
            cnames[31] = "Essex";
            cnames[113] = "Warren";
            cnames[19] = "Clinton";
            cnames[115] = "Washington";
            cnames[83] = "Rensselaer";
            cnames[21] = "Columbia";
            cnames[27] = "Dutchess";
            cnames[91] = "Saratoga";
            cnames[35] = "Fulton";
            cnames[93] = "Schenectady";
            cnames[57] = "Montgomery";
            cnames[25] = "Delaware";
            cnames[77] = "Otsego";
            cnames[65] = "Oneida";
            cnames[53] = "Madison";
            cnames[21] = "Columbia";
            cnames[27] = "Dutchess";
            cnames[79] = "Putnam";
            cnames[119] = "Westchester";
            cnames[105] = "Sullivan";
            cnames[71] = "Orange";
            cnames[111] = "Ulster";
            cnames[39] = "Greene";
            cnames[95] = "Schoharie";
            cnames[1] = "Albany";
            cnames[87] = "Rockland";
            cnames[103] = "Suffolk";
            cnames[59] = "Nassau";
            cnames[81] = "Queens";
            cnames[85] = "Richmond";
            cnames[47] = "Kings";
            cnames[5] = "Bronx";
            cnames[61] = "New York";

            for(liter = myTracts.begin(); liter != myTracts.end();
                    liter++){
                c = (*liter)->getMyCounty();
                pz[c] += (*liter)->getArea();
            }

            for(miter = pz.begin(); miter != pz.end(); miter++){
                frac =
                    (double)(miter->second)/(double)((miter->first)->getArea());
                cout <<
                    cnames[((miter->first)->getTractList()).front()->getCounty()]
                    << " " << frac << " ";
                retval += frac*frac;
            }
            cout << endl;
            return M*(retval);
        }

        OGRPoint *centerOfMass(){
            list<Tract *>::iterator liter;
            double x=0,y=0;

            for(liter = myTracts.begin(); liter != myTracts.end();
                    liter++){
                x += ((*liter)->getPop())*((*liter)->getCentroid())->getX();
                y +=
                    ((*liter)->getPop())*((*liter)->getCentroid())->getY();
            }
            x = x/getPop();
            y = y/getPop();

            OGRPoint *retval = new OGRPoint();
            retval->setX(x);
            retval->setY(y);
            return retval;
        }
        double bcBB(){
            list<Tract *>::iterator perim;
            list<Tract *> p = myTracts;
            double minX = 999999999999.;
            double minY = 999999999999.;
            double maxX = -999999999999.;
            double maxY = -999999999999.;
            double curX;
            double curY;
            OGRPoint *pt;

            for(perim = p.begin(); perim != p.end(); perim++){
                pt = (*perim)->getCentroid();
                curX = pt->getX();
                curY = pt->getY();
                if(curX < minX){
                    minX = curX;
                }
                if(curY < minY){
                    minY = curY;
                }
                if(curX > maxX){
                    maxX = curX;
                }
                if(curY > maxY){
                    maxY = curY;
                }
            }

            return 4.*pow(maxX-minX+(maxY-minY),2);
        }

        vector<District *> whatBordersMe(){
            map<District *,bool>seenit;
            list<Tract *>::iterator perim;
            list<Tract *> p = getFrontier();
            vector<District *> retval;
            for(perim = p.begin(); perim != p.end(); perim++){
                if(!seenit[(*perim)->getDistrict()]){
                    seenit[(*perim)->getDistrict()] = true;
                    retval.push_back((*perim)->getDistrict());
                }
            }

            return retval;
        }

        vector<Tract *> sharesBorder(District *d){
            list<Tract *>::iterator perim;
            list<Tract *> p = getFrontier();
            vector<Tract *> retval;
            for(perim = p.begin(); perim != p.end(); perim++){
                if((*perim)->getDistrict() == d){
                    retval.push_back(*perim);
                }
            }

            return retval;
        }

```

```

    double compactScore(){
        //return getIsoPerim();
        //double M = .1;
        //return M*getArea()/bcBB();
        /*
        double M = -10000;
        list<Tract *>::iterator perim;
        list<Tract *> p = myTracts;
        double avgDist = 0;
        OGRPoint *c = centerOfMass();
        for(perim = p.begin(); perim != p.end(); perim++){
            avgDist += (*perim)->dist(c);
        }
        avgDist = avgDist/p.size();
        double retval = 0;
        for(perim = p.begin(); perim != p.end(); perim++){
            retval = pow((1-(*perim)->dist(c)/avgDist),2.);
        }
        delete c;
        return M*retval/(p.size()-1);
        */

        double M = 30;
        list<Tract *>::iterator liter;
        vector<Tract *> n;
        int i;
        double count = 0;
        for(liter = myTracts.begin(); liter != myTracts.end();
                liter++){
            n = (*liter)->getN();
            for(i=0; i < n.size(); i++){
                if(n[i]->getDistrict() == this){
                    count++;
                }
            }
        }
        return count * M /
            (((double)myTracts.size())*((double)myTracts.size() - 1));
        //
       /*
        double M = 10.;
        list<Tract *> p = getPerimeter();
        double b = (double)p.size();
        int nt = getNumTracts();
        return M*((double)nt)/pow(b+4.,2.);*/
    }

    inline double varScore(){
        double M = -1000.;
        return M*(getPop() -
                AVGPEOPLE)*(1./getPop())*(getPop()-AVGPEOPLE)*(1./getPop());
    }

    map<Tract *,bool> visited;

    bool isContiguous(){
        if(this == BLANKDIST){
            return true;
        }
        list <Tract *>::iterator liter;
        visited.clear();
        dfs(getTractList().front());
        bool visitedall = true;
        for(liter = myTracts.begin(); liter != myTracts.end();
                liter++){
            if(visited[(*liter)] == false){
                visitedall = false;
```

```
                    break;
                }
            }

            return visitedall;
        }

        void dfs(Tract *t){
            visited[t] = true;
            vector <Tract *> n = t->getN();
            int i;
            for(i=0; i < n.size(); i++){
                if((this == n[i]->getDistrict()) && (!visited[n[i]])){
                    dfs(n[i]);
                }
            }
        }

        double getValue(){
            if(this == BLANKDIST){
                return 0;
            }
            double M=10000;
            double p = (double)getPop();
            if(p < AVGPEOPLE){
                return M*sqrt(p/AVGPEOPLE);
            }
            return M-4*M*((p-AVGPEOPLE)/p)*((p-AVGPEOPLE)/p);
        }

        // perimeter -> set of nodes that are in this and border
        // something not in this
        list<Tract *> getPerimeter(){
            // go through all the Tracts...
            list<Tract *>::iterator liter;
            list<Tract *> returnval;
            for(liter = myTracts.begin(); liter != myTracts.end();
                    liter++){
                if((*liter)->onPerimeter()){
                    returnval.push_front(*liter);
                }
            }
            return returnval;
        }


        // frontier -> set of nodes that border this
        list<Tract *> getFrontier(){
            // go thru all the vectors
            // add to master list only if it's not == this
            list<Tract *>::iterator liter;
            list<Tract *> returnval;
            map<Tract *,bool> seenit;

            vector<Tract *> v;
            int i;
            for(liter = myTracts.begin(); liter != myTracts.end();
                    liter++){
                v = (*liter)->getN();
                for(i=0; i < v.size(); i++){
                    if((this != v[i]->getDistrict()) && !seenit[v[i]]){
                        returnval.push_front(v[i]);
                        seenit[v[i]] = true;
                    }
                }
            }

            //returnval.sort();
            /*
            Tract *prev;
            if(returnval.size() > 1){
                prev = returnval.front();
                for(liter = ((returnval.begin())++); liter != returnval.end();
                        liter++){
                        if(prev == (*liter)){
                            returnval.remove(prev);
                            addin.push_back(prev);
                        }
                        prev = *liter;
                }
            }
            for(i=0; i < addin.size(); i++){
                returnval.push_front(addin[i]);
            }*/
            //returnval.unique();
            return returnval;
        }
/*
        bool minmex(const Tract* a,const Tract *b){
            return ((a.getCentroid())->getX() <
                    (b.getCentroid())->getX());
        }

        double *getMinMaxX(){
            double returnval[2];
            myTracts.sort(minmex);
            returnval[0] = (myTracts.front()->getCentroid())->getX();
            returnval[1] = (myTracts.back()->getCentroid())->getX();
            return returnval;
        }

        bool minmey(const Tract* a, const Tract *b){
            return ((a.getCentroid())->getY() <
                    (b.getCentroid())->getY());
        }

        double *getMinMaxY(){
            double returnval[2];
            myTracts.sort(minmey);
            returnval[0] = (myTracts.front()->getCentroid())->getY();
            returnval[1] = (myTracts.back()->getCentroid())->getY();
            return returnval;
        }

        list<Tract *> cleavelessthanx(double target){
            list<Tract *> returnval;
            list<Tract *>::iterator iter;
            for(iter=myTracts.start();iter != myTracks.end(); iter++){
                if(((*iter)->getCentroid())->getX() < target){
                    myTracts.remove(*iter);
                    returnval.push_back(*iter);
                }
            }
        }*/
};

#endif
```

```

// Allocation.h - header file for an Allocation
// an Allocation consists of an array of districts (29) and a heuristic
// value.

#ifndef ALLOCATION_H
#define ALLOCATION_H

#include <iostream>
```

```cpp
#include <cmath>
#include "District.h"

using namespace std;

class Allocation {
    protected:
        District* d[29];

    public:
        Allocation(){ }
        Allocation(District **ds){
            int i;
            for(i=0; i < 29; i++){
                d[i] = ds[i];
            }
        }

        District **getDistricts(){
            return d;
        }
};

#endif
```

```cpp


#include "ogrsf_frmts.h"
#include <iostream>
#include <fstream>
#include <iomanip>
#include <string>
#include <map>
#include "Tract.h"
#include "County.h"
#include "District.h"
#include "Allocation.h"
//#include "rng.h"
#include <sstream>
#include <cstdlib>
#include <ctime>
#include <vector>
#include "Fnode.h"
#include <algorithm>

const int NTRACT = 4907;
const int NDIST = 29;
const double AVGPEOPLE = 18976457./(float)NDIST;
const int NCOUNTY = 62;
//const int NLEVELS = 20;
District *BLANKDIST;
const bool PRINTHEU = false;

using namespace std;

void plotAllocation(Allocation *a,string fname);
District **getNeighbor(District **d, Tract** allTracts, double
        **distmat);
void moveTract(Tract *t, District *newd);
double getBadness(District **d,double **distmat);
void clarify(Tract **allTracts);
void addneighrecur(Tract *t,District *changeto,District *background,int
        levels);
double generateScore(District **d, County **allCounties);
vector <Fnode *> unionFrontier(District **d);
double getBC(vector<Tract *> startingpoints,Tract *t);

bool compf(Fnode *lhs, Fnode *rhs){
    // greater than, not less than, b/c we want to sort descending
    return lhs->getScore() > rhs->getScore();
}

bool eqf(Fnode *lhs, Fnode *rhs){
    return lhs->getScore() == rhs->getScore();
}

bool eq_func(Tract *lhs, Tract *rhs){
    return lhs == rhs;
}
bool compbefore(Fnode *lhs, Fnode *rhs){
    if(rhs->getTract() >= lhs->getTract()){
        return true;
    } else if(rhs->getTract() == lhs->getTract()){
        if(rhs->getDistrict() >= lhs->getDistrict()){
            return true;
        }
    }
    return false;
}

bool eqbefore(Fnode *lhs, Fnode *rhs){
    return((rhs->getTract() == lhs->getTract()) &&
            (rhs->getDistrict() == lhs->getDistrict()));
}

bool comp_func(Tract *lhs, Tract *rhs){
    return lhs < rhs;
}

string inttostring(const int i){
    ostringstream stream;
    stream << i;
    return stream.str();
}

double randdub(){
    return rand()/(double)RAND_MAX;
}
//returns between lo and hi inclusive
int randint(int low, int high){
    return(low+(int)floor(randdub()*(high-low+1)));
}

vector <Tract *>copyvec(const vector<Tract *> &in){
    int i;
    vector <Tract *> returnval;
    for(i=0;i<in.size();i++){
        returnval[i] = in[i];
    }
}

int main(int argc, char * const argv[]) {
    srand((unsigned)time(NULL));

    OGRRegisterAll();

    OGRDataSource *myfile;

    myfile = OGRSFDriverRegistrar::Open("./polygons/", FALSE);
    if(myfile == NULL){
        cerr << "Can't open file" << endl;
        return 1;
    }
    cout << "Opened file appropriately!" << endl;
    cout << "File has " << myfile->GetLayerCount() << " layers" << endl;

    OGRLayer *layer = myfile->GetLayer(0);
    if(!layer){
        cerr << "Cannot open layer" << endl;
        return 1;
    }

    cout << "Layer has " << layer->GetFeatureCount() << " features" <<
        endl;
    int numtracts = layer->GetFeatureCount();
    int i,j;
    OGRFeature *feat;
    int populationindex;
    int totalpop = 0;
    map<string,int> IDtoIref;
    map<int,int> CkeytoRkey; // county key in file to our real keys.
    Tract *allTracts[NTRACT];
    bool **bmat = new bool*[NTRACT];
    double **distmat = new double*[NTRACT];

    double pdscore,pcscore,fdscore,fcscore;
    Allocation *a;
    County **allCounties = new County*[NCOUNTY];
    for(i=0; i < NCOUNTY; i++){
        allCounties[i] = new County();
    }

    int cindex=-1;
    for(i=0; i < numtracts; i++){
        feat = layer->GetNextFeature();
        if(!feat){
            cerr << "Could not read feature, exiting!" << endl;
            return 1;
        }
        allTracts[i] = new Tract(feat,i);
        IDtoIref[allTracts[i]->getID()] = i;
        // Link to counties...
        if(CkeytoRkey.count(allTracts[i]->getCounty()) == 0){
            cindex++;
            CkeytoRkey[allTracts[i]->getCounty()] = cindex;
        }
        allCounties[CkeytoRkey[allTracts[i]->getCounty()]]->addToCounty(allTracts[i]);

        delete feat;
        feat = NULL;
    }

    cout << "beginning to read border file..." << endl;
    ifstream bo;
    bo.open("border.txt");
    for(i=0; i < NTRACT; i++){
        bmat[i] = new bool[NTRACT];
        for(j=0; j < NTRACT; j++){
            bo >> bmat[i][j];
        }
    }
    bo.close();
    cout << "finished reading border file" << endl;

    vector <Tract *> n;
    for(i=0; i < NTRACT; i++){
        for(j=0; j < NTRACT; j++){
            if(bmat[i][j]){
                n.push_back(allTracts[j]);
            }
        }
        allTracts[i]->setN(n);
        n.clear();
    }

    cout << "beginning calculating centroid distances" << endl;
    for(i=0; i < NTRACT; i++){
        distmat[i] = new double[NTRACT];
        for(j=0; j < NTRACT; j++){
            if(j < i){
                distmat[i][j] = distmat[j][i];
            } else {
                distmat[i][j] = allTracts[i]->distC(allTracts[j]);
            }
        }
    }
    cout << "finished calculating centroid distances" << endl;

    District *d[NDIST+1]; // d[NDIST] = blank canvas....
    for(i=0; i < NDIST+1; i++){
        d[i] = new District();
    }

    BLANKDIST = d[NDIST];

    // initially we paint everything NDIST...
    for(i=0; i < NTRACT; i++){
        allTracts[i]->setDistrict(BLANKDIST);
        BLANKDIST->addToDistrict(allTracts[i]);
    }


    string spoint = "     3483864"; // remember the spaces!
    int iref = IDtoIref[spoint];

    if(!allTracts[iref]){
        cerr << "Could not find starting node, exiting!" << endl;
        return 1;
    }

    // color it!
    District *curd;
    //curd = d[0];
    //moveTract(allTracts[iref],curd);
    // in each step, get list of possible frontier nodes.
    // find the value of adding each node.
    // add the one with highest value only if the new value is increased
    District *checkme;
    list <Tract *> f;
    list <Tract *>::iterator liter;
    double hiscore;
    Tract *addme;
    double curval,tmpscore;
    bool done;
    hiscore = -999999999;
    addme = NULL;
    Tract *abba;
    vector <Tract *> startingpoints;
    double maxdist;
    Tract *thevest; // "vest is best!"
    abba = allTracts[iref];
    /* distance maximin
    startingpoints.push_back(abba);
    for(i=1; i < NDIST; i++){
        maxdist = -1.;
        for(j=0; j < NTRACT; j++){
            tmpscore=getBC(startingpoints,allTracts[j]);
            if(tmpscore > maxdist){
                maxdist = tmpscore;
                thevest = allTracts[j];
            }
        }
        startingpoints.push_back(thevest);
    }
    for(i=0; i < NDIST; i++){
        abba = startingpoints[i];
        moveTract(abba,d[i]);
    }*/
    bool flag;

    cout << "Allocating initial random districts" << endl;
    for(i=0; i <NDIST; i++){
        flag = false;
        do {
            j = randint(0,NTRACT-1);
            abba = allTracts[j];
            if(randdub() < (double)abba->getPop()/25000.)
                flag = true;
        } while(abba->getDistrict() != BLANKDIST || !flag);
        moveTract(abba,d[i]);
    }
    cout << "Done random allocation" << endl;
    a = new Allocation(d);
    plotAllocation(a, "initial");
    vector<Fnode *> curfr;
    Fnode *best;
    County *iq;

    while((BLANKDIST->getTractList()).size() > 0){
        curfr = unionFrontier(d);
        //sort(curfr.begin(),curfr.end(),compbefore);
        //curfr.erase(unique(curfr.begin(),curfr.end(),eqbefore),curfr.end());
        cout << "Current size is: " <<
            (BLANKDIST->getTractList()).size() <<
            " Frontier: " << curfr.size() << endl;
        for(i=0; i < curfr.size(); i++){
            pdscore = (curfr[i]->getDistrict())->getValue();
            iq =
                allCounties[CkeytoRkey[(curfr[i]->getTract())->getCounty()]];
            pcscore = iq->getValue();
            moveTract(curfr[i]->getTract(),curfr[i]->getDistrict());
            fdscore = (curfr[i]->getDistrict())->getValue();
            fcscore = iq->getValue();
            //tmpscore = generateScore(d,allCounties);
            tmpscore = fdscore+fcscore-pdscore-pcscore;
            // methodology: generate scores for all, sort, take the top
            // ceil(1/50th) of points.
            curfr[i]->setScore(tmpscore);
            if(tmpscore >= hiscore){
                hiscore = tmpscore;
                best = curfr[i];
            }
            moveTract(curfr[i]->getTract(),BLANKDIST);
        }
        // sort descending scores here
        sort(curfr.begin(),curfr.end(),compf);
        //curfr.erase(unique(curfr.begin(),curfr.end(),eqf),curfr.end());
        // do the movements;

        j = (int)floor((double)curfr.size()/30.);
        for(i=j;i != -1 ; i--){
            moveTract(curfr[i]->getTract(),curfr[i]->getDistrict());
        }
        curfr.clear();
        //moveTract(best->getTract(),best->getDistrict());
        cout << "Score: " << generateScore(d,allCounties) << endl;
    }

    // District-by-District
    /*
    double pdoth,fdoth;
    bool flag=false;
    for(i=0; i < NDIST; i++){
        curd = d[i];
        flag = false;
        do {
            j = randint(0,NTRACT-1);
            abba = allTracts[j];
            if(randdub() < (double)abba->getPop()/25000.)
                flag = true;
        } while(abba->getDistrict() != BLANKDIST || !flag);
        addme = abba;

        cout << "Starting District " << i+1 << endl;
        /*
        curd = d[i];
        f = BLANKDIST->getTractList();
        for(liter = f.begin(); liter != f.end(); liter++){
            moveTract(*liter,curd);
            tmpscore = generateScore(d,allCounties);
            if(tmpscore >= hiscore){
                hiscore = tmpscore;
                addme = *liter;
            }
            moveTract(*liter,BLANKDIST);
        }*
        moveTract(addme,curd);
        done = false;
        while(!done){
            curval = generateScore(d,allCounties);
            cout << "Score: " << curval << endl;
            hiscore = -50.;
            addme = NULL;
            f = curd->getFrontier();
            //cout << "Frontier has " << f.size() << " tracts" << endl;
            for(liter = f.begin(); liter != f.end(); liter++){
                // add liter to current allocation, getvalue, check and
                // unwind, settign hiscore and addme if necessary.
                checkme = (*liter)->getDistrict();
                if(checkme == curd){
                    cerr << "There is a problem with frontier generation!"
                        << endl;
                }
                if(checkme->isContiguous()){
                    pdoth = checkme->getValue();
                    pdscore = curd->getValue();
                    iq =
                        allCounties[CkeytoRkey[(*liter)->getCounty()]];
                    pcscore = iq->getValue();
                    moveTract(*liter,curd);
                    fdoth = checkme->getValue();
                    fdscore = curd->getValue();
                    fcscore = iq->getValue();
                    //tmpscore = generateScore(d,allCounties);
                    tmpscore = fdscore+fcscore+fdoth - pcscore - pdscore
                        - pdoth;
                    if(tmpscore >= hiscore){
                        addme = *liter;
                        hiscore = tmpscore;
                    }
                    moveTract(*liter,checkme);
                }
            }
            if(addme == NULL){
                done = true;
            } else {
                moveTract(addme,curd);
            }
        }
    }
//*/
/*
    for(i=0; i < NTRACT; i++){
        allTracts[i]->setDistrict(d[0]);
        d[0]->addToDistrict(allTracts[i]);
    }
    int seedind,k;
    cout << "Beginning recursive initial districting" << endl;
    for(i=1; i < NDIST; i++){
        do {
            seedind = randint(0,NTRACT-1);
        } while(allTracts[seedind]->getDistrict() != d[0]);

        // seed with self, neighbors, neighbors of neighbors
        addneighrecur(allTracts[seedind],d[i],d[0],NLEVELS);
        if((allTracts[seedind]->getN()).front()->getDistrict() !=
                allTracts[seedind]->getDistrict()){
            moveTract((allTracts[seedind]->getN()).front(),allTracts[seedind]->getDistrict()
                );
        }
    }
    // add District 0 possible elimination

    *
    for(i=1; i < NDIST; i++){
        if(thechosen[i]->getDistrict() != d[i]){
            moveTract(thechosen[i],d[i]);
        }
    }*

    for(i=0; i < NDIST; i++){
        cout << "District " << i+1 << ": " << d[i]->getPop() << endl;
        cout << " has " << (d[i]->getTractList()).size() << endl;
    }


    Allocation *a = new Allocation(d);
    plotAllocation(a, "initial");
    District **maybe;
    District **curr = d;
    for(i=0; i< 1000; i++){
        //if(!(i%10))
        clarify(allTracts);
        cout << "Step " << i << " badness: " << getBadness(curr,distmat)
            << endl;
        maybe = getNeighbor(curr,allTracts,distmat);

        if(!maybe){
            //cout << "I didn't improve!" << endl;
        } else {
            curr = maybe;
        }
    }*/

    int sumpump=0;
    for(i=0; i < NDIST+1; i++){
        sumpump += d[i]->getPop();
        cout << "District " << i+1 << ": " << d[i]->getPop() << endl;
    }

    if(argc == 2){
        list <Tract *>doolist;
        list <Tract *>::iterator liter;
        ofstream outfile(argv[1]);
        double variance=0;
        for(i=0; i < NDIST; i++){
            variance += pow((double)(d[i]->getPop()-AVGPEOPLE),2.);
        }
        variance = sqrt(variance);
        outfile << variance << endl;
        outfile << generateScore(d,allCounties) << endl;
        for(i=0; i < NDIST; i++){
            outfile << "D ";
            doolist = d[i]->getTractList();
            for(liter=doolist.begin(); liter != doolist.end(); liter++){
                outfile << (*liter)->getIndex() << " ";
            }
            outfile << endl;
        }
        outfile.close();
    }

    cout << "Total population: " << sumpump << endl;
    a = new Allocation(d);
    plotAllocation(a, "testing");
    return 0;
}

// measures bc metric, returns max found...
double getBC(vector<Tract *> startingpoints,Tract *t){
    int i;
    double minv = 999999999999999999.;
    double tmp;
    for(i=0; i < startingpoints.size(); i++){
        tmp = t->bcMetric(startingpoints[i]);
        if(tmp < minv){
            minv = tmp;
        }
    }

    return minv;
}

vector <Fnode *> unionFrontier(District **d){
    int i,j,k;
    list<Tract *> f;
    list<Tract *>::iterator liter,jiter,kiter;
    Fnode *tmp;
    bool flag;
    vector <Fnode *> retval;
    for(i=0; i < NDIST; i++){
        f = d[i]->getFrontier();
        /*
        for(jiter = f.begin(); jiter != f.end(); jiter++){
            flag = false;
            for(kiter = jiter; kiter != f.end(); kiter++){
                if(((*jiter) == (*kiter)) && !flag){
                    flag = true;
                } else if((*jiter) == (*kiter)){
                    cout << "Duplicate in the frontier!" << endl;
                }
            }
        }
        */
        for(liter = f.begin(); liter != f.end(); liter++){
            if((*liter)->getDistrict() == BLANKDIST){
                tmp = new Fnode(*liter,d[i]);
                retval.push_back(tmp);
            }
        }
    }

    return retval;
}

```

```
double generateScore(District **d, County **allCounties){
    int i;
    double pval=0;
    double cval=0;

    for(i=0; i < NDIST; i++){
        pval += d[i]->getValue();
    }

    for(i=0; i < NCOUNTY; i++){
        cval += allCounties[i]->getValue();
    }

    if(PRINTHEU)
        cout << "Population Score: " << pval << " County Score: " << cval << endl;
    return pval+cval;
}

void addneighrecur(Tract *t,District *changeto,District *background,int
        levels){
    if(levels == 0)
        return;
    if(t->getDistrict() == changeto || t->getDistrict() != background)
        return;

        moveTract(t,changeto);
        vector <Tract *> nvec;;
        int j;
        nvec = t->getN();
        for(j=0; j < nvec.size(); j++){
            addneighrecur(nvec[j],changeto,background,levels-1);
        }
}

void clarify(Tract **allTracts){
    int i,j;
    // if everything around me is another color, then I change
    District *me,*oth;
    vector <Tract *> n;
    bool changeme;
    for(i=0; i < NTRACT; i++){
        me = allTracts[i]->getDistrict();
        if(me->getTractList().size() <= 2){
            continue;
        }
        n = allTracts[i]->getN();
        if(n.size() > 0){
            changeme = true;
            for(j=0; j < n.size(); j++){
                if(me == n[j]->getDistrict()){
                    changeme = false;
                    break;
                }
            }
            /*
            oth = n[0]->getDistrict();
            if(oth != me){
                changeme = true;
                for(j=1; j < n.size(); j++){
                    if(oth != n[j]->getDistrict()){
                        changeme = false;
                        break;
                    }
                }
            }*/
            if(changeme){
                oth = n[randint(0,n.size()-1)]->getDistrict();
                cout << "Found enclave!" << endl;
                moveTract(allTracts[i],oth);
                changeme = false;
            }
        }
    }
}


District **getNeighbor(District **d, Tract** allTracts, double **distmat){
    double curval = getBadness(d,distmat);
    int i;

    bool done = false;
    Tract *tmp,*posc;
    vector <Tract *> in;
    while(!done){
        tmp = allTracts[randint(0,NTRACT-1)];
        if(!tmp->onPerimeter()){
            continue;
        } else {
            in = tmp->getN();
            posc = in[randint(0,in.size()-1)];
            if(posc->getDistrict() != tmp->getDistrict())
                done = true;
        }
    }
    /*
    vector <Tract *> borders;

    for(i=0;i < NTRACT; i++){
        if(allTracts[i]->onPerimeter()){
            borders.push_back(allTracts[i]);
        }
    }

    while(!done){
        tmp = borders[randint(0,borders.size()-1)];
        in = tmp->getN();
        posc = in[randint(0,in.size()-1)];
        if(posc->getDistrict() != tmp->getDistrict())
            done = true;
    }
    */
    District *oldd = tmp->getDistrict();
    District *newd = posc->getDistrict();
    double movet;
    double movec;
    double swap;

    // option one: let's move tmp to newd:

    moveTract(tmp,newd);
    movet = getBadness(d,distmat);
    // huh. That didn't work. Let's try the other way...
    moveTract(tmp,oldd);
    moveTract(posc,oldd);
    movec = getBadness(d,distmat);

    // Try the swap...
    moveTract(tmp,newd);
    swap = getBadness(d,distmat);

    list <double> l;
    l.push_front(curval);
    l.push_front(movet);
    l.push_front(movec);
    l.push_front(swap);

    l.sort();
    // current state: swapped
    if(l.front() == curval){
        moveTract(tmp,oldd);
        moveTract(posc,newd);
        return NULL;
    } else if(l.front() == movet){
        moveTract(posc,newd);
        return d;
    } else if(l.front() == movec){
        moveTract(tmp,oldd);
        return d;
    } else {
        return d;
    }
}

// house cleaning to keep data structs in order
void moveTract(Tract *t, District *newd){
    District *oldd = t->getDistrict();
    if(oldd == newd){
        cerr << "Trying to change to already fixed district!" << endl;
        return;
    }
    list <Tract *> l = oldd->getTractList();
    l.remove(t);
    l = newd->getTractList();
    l.push_front(t);
    t->setDistrict(newd);
    oldd->removeFromDistrict(t);
    newd->addToDistrict(t);
}

double getBadness(District **d,double **distmat){
    int i;
    double sum=0;

    // Linf norm (max)
    /*
    for(i=0; i < NDIST; i++){
        if(d[i]->getPop() > sum){
            sum = d[i]->getPop();
        }
    }*/
    // L2 norm (variance):

    for(i=0; i < NDIST; i++){
        sum += pow(d[i]->getPop()-AVGPEOPLE,2);
    }
    sum = sqrt(sum); // add constant factor here at some point


    double dist=0;
    list <Tract *>lind;
    list <Tract *>::iterator iti;
    list <Tract *>::iterator jtj;
    double mydist=0;
    for(i=0; i < NDIST; i++){
        lind = d[i]->getTractList();
        for(iti = lind.begin(); iti != lind.end(); iti++){
            for(jtj = iti; jtj != lind.end(); jtj++){
                mydist += distmat[(*iti)->getIndex()][(*jtj)->getIndex()];
            }
        }
        dist +=
            mydist/(lind.size()*(lind.size()-1)*sqrt(d[i]->getArea()));
        mydist = 0;
    }

    dist = dist * 700000;
    cout << "Sum of Distances Metric: " << dist << " Population Metric: " << sum << endl;
    return dist+sum;
}


void plotAllocation(Allocation *a,string fname){
    // plots an Allocation to a file

    const char *pszDriverName = "ESRI Shapefile";
    OGRSFDriver *poDriver;

    OGRRegisterAll();

    poDriver =
        OGRSFDriverRegistrar::GetRegistrar()->GetDriverByName(
                pszDriverName);
    if(!poDriver){
        cerr << "Could not initialize driver for writing!" << endl;
        return;
    }

    OGRDataSource *poDS;
    OGRLayer *layer;
    District **d = a->getDistricts();
    int i;
    string curname,lname;
    OGRFeature *tmpf;
    list <Tract *>tracts;
    list <Tract *>::iterator iter;

    for(i=0; i < NDIST; i++){
        tracts = d[i]->getTractList();
        curname = fname + inttostring(i) + ".shp";
        poDS = poDriver->CreateDataSource(fname.c_str(), NULL);
        if(!poDS){
            cerr << "Could not create output file!" << endl;
            return;
        }
        lname = "District " + inttostring(i+1);
        layer = poDS->CreateLayer(lname.c_str(), NULL, wkbUnknown,NULL);
        if(!layer){
            cerr << "Layer creation failed!" << endl;
            return;
        }

        for(iter = tracts.begin(); iter != tracts.end(); iter++){
            tmpf = new OGRFeature(layer->GetLayerDefn());
            tmpf->SetGeometry((*iter)->getGeo());
            if(layer->CreateFeature(tmpf) != OGRERR_NONE){
                cerr << "Could not create feature!" << endl;
                return;
            }
            OGRFeature::DestroyFeature(tmpf);
        }
        OGRDataSource::DestroyDataSource(poDS);
    }
}
```

```
#include "ogrsf_frmts.h"
#include <iostream>
#include <fstream>
#include <iomanip>
#include <string>
#include <map>
#include "Tract.h"
//#include "County.h"
//#include "District.h"
```

```
#include "Allocation.h"
//#include "rng.h"
#include <sstream>
#include <cstdlib>
#include <ctime>
#include <vector>
#include "Fnode.h"
#include <algorithm>

const int NTRACT = 4907;
const int NDIST = 29;
const double AVGPEOPLE = 18976457./(float)NDIST;
const int NCOUNTY = 62;
//const int NLEVELS = 20;
District *BLANKDIST;
const bool PRINTHEU = false;

using namespace std;

void plotAllocation(Allocation *a,string fname);
void moveTract(Tract *t, District *newd);
vector <Fnode *> unionFrontier(District **d);
double getBC(vector<Tract *> startingpoints,Tract *t);
District *largestD(District **d);
double partTwoScore(District **d,County **allCounties);
District *smallestD(District **d);
vector <Fnode *> addingMoves(District *dis);
vector <Fnode *> reducingMoves(District *dis);
District *nextD(District **d);

bool compf(Fnode *lhs, Fnode *rhs){
    // greater than, not less than, b/c we want to sort descending
    return lhs->getScore() > rhs->getScore();
}

bool eqf(Fnode *lhs, Fnode *rhs){
    return lhs->getScore() == rhs->getScore();
}

bool eq_func(Tract *lhs, Tract *rhs){
    return lhs == rhs;
}
bool compbefore(Fnode *lhs, Fnode *rhs){
    if(rhs->getTract() >= lhs->getTract()){
        return true;
    } else if(rhs->getTract() == lhs->getTract()){
        if(rhs->getDistrict() >= lhs->getDistrict()){
            return true;
        }
    }
    return false;
}

bool eqbefore(Fnode *lhs, Fnode *rhs){
    return((rhs->getTract() == lhs->getTract()) &&
            (rhs->getDistrict() == lhs->getDistrict()));
}

bool comp_func(Tract *lhs, Tract *rhs){
    return lhs < rhs;
}

string inttostring(const int i){
    ostringstream stream;
    stream << i;
    return stream.str();
}

```

```
double randdub(){
    return rand()/(double)RAND_MAX;
}
//returns between lo and hi inclusive
int randint(int low, int high){
    return(low+(int)floor(randdub()*(high-low+1)));
}

vector <Tract *>copyvec(const vector<Tract *> &in){
    int i;
    vector <Tract *> returnval;
    for(i=0;i<in.size();i++){
        returnval[i] = in[i];
    }
}

int idFromString(char *s,map<string,int> m){
    string k(s);

    return m[k];
}

int main(int argc, char * const argv[]) {
    srand((unsigned)time(NULL));

    OGRRegisterAll();

    OGRDataSource *myfile;

    myfile = OGRSFDriverRegistrar::Open("./polygons/", FALSE);
    if(myfile == NULL){
        cerr << "Can't open file" << endl;
        return 1;
    }
    cout << "Opened file appropriately!" << endl;
    cout << "File has " << myfile->GetLayerCount() << " layers" << endl;

    OGRLayer *layer = myfile->GetLayer(0);
    if(!layer){
        cerr << "Cannot open layer" << endl;
        return 1;
    }

    cout << "Layer has " << layer->GetFeatureCount() << " features" <<
        endl;
    int numtracts = layer->GetFeatureCount();
    int i,j;
    OGRFeature *feat;
    int populationindex;
    int totalpop = 0;
    map<string,int> IDtoIref;
    map<int,int> CkeytoRkey; // county key in file to our real keys.
    Tract *allTracts[NTRACT];
    bool **bmat = new bool*[NTRACT];
    double **distmat = new double*[NTRACT];

    Allocation *a;
    County **allCounties = new County*[NCOUNTY];
    for(i=0; i < NCOUNTY; i++){
        allCounties[i] = new County();
    }

    int cindex=-1;
    for(i=0; i < numtracts; i++){
        feat = layer->GetNextFeature();
        if(!feat){
            cerr << "Could not read feature, exiting!" << endl;
            return 1;
        }
        allTracts[i] = new Tract(feat,i);
        IDtoIref[allTracts[i]->getID()] = i;
        // Link to counties...
        if(CkeytoRkey.count(allTracts[i]->getCounty()) == 0){
            cindex++;
            CkeytoRkey[allTracts[i]->getCounty()] = cindex;
        }
        allCounties[CkeytoRkey[allTracts[i]->getCounty()]]->addToCounty(allTracts[i]);
        allTracts[i]->setCounty(allCounties[CkeytoRkey[allTracts[i]->getCounty()]]);

        delete feat;
        feat = NULL;
    }

    cout << "beginning␣to␣read␣border␣file..." << endl;
    ifstream bo;
    bo.open("border.txt");
    for(i=0; i < NTRACT; i++){
        bmat[i] = new bool[NTRACT];
        for(j=0; j < NTRACT; j++){
            bo >> bmat[i][j];
        }
    }
    bo.close();
    cout << "finished␣reading␣border␣file" << endl;

  /*
    cout << "beginning calculating centroid distances" << endl;
    for(i=0; i < NTRACT; i++){
        distmat[i] = new double[NTRACT];
        for(j=0; j < NTRACT; j++){
            if(j < i){
                distmat[i][j] = distmat[j][i];
            } else {
                distmat[i][j] = allTracts[i]->distC(allTracts[j]);
            }
        }
    }
    cout << "finished calculating centroid distances" << endl;
*/
    District *d[NDIST+1]; // d[NDIST] = blank canvas....
    for(i=0; i < NDIST+1; i++){
        d[i] = new District();
    }

    BLANKDIST = d[NDIST];
    // Read in file here....
    cout << "opening␣input␣file...." << endl;
    if(argc >= 2){
        list <Tract *>doolist;
        list <Tract *>::iterator liter;
        ifstream infile(argv[1]);
        if(!infile){
                cerr << "Could␣not␣open␣" << argv[1] << endl;
                return 1;
        }
        double upp;
        infile >> upp;
        cout << "Variance:␣" << upp << endl;
        infile >> upp;
        cout << "Score:␣" << upp << endl;
        int inp;
        for(i=-1; (i < NDIST) && !infile.eof(); i++){
            infile >> inp;
            while((inp != -1) && !infile.eof()){
                allTracts[inp]->setDistrict(d[i]);
                d[i]->addToDistrict(allTracts[inp]);
                infile >> inp;
            }
        }

        infile.close();
    } else {
            cerr << "Must call an input file..." << endl;
            return 1;
    }


    double** intermat = new double*[NTRACT];
    double *myp = new double[NTRACT];
    OGRGeometry *ia;
    OGRGeometry *ib;
    OGRGeometry *u;
    double sz;
    for(i=0; i < NTRACT; i++){
        ia = allTracts[i]->getGeo();
        myp[i] = (((OGRPolygon *)ia)->getExteriorRing())->get_Length();
    }

    /*
    for(i=0; i < NTRACT; i++){
        intermat[i] = new double[NTRACT];
        for(j=0; j < NTRACT; j++){
            if(!bmat[i][j]){
                intermat[i][j]=0;
                continue;
            }

            if(i > j){
                intermat[i][j] = intermat[j][i];
                continue;
            }
            ia = allTracts[i]->getGeo();
            ib = allTracts[j]->getGeo();
            u = ia->Union(ib);
            sz = (((OGRPolygon *)u)->getExteriorRing())->get_Length();
            intermat[i][j] = (double)(myp[i]+myp[j]-sz)/(double)2.;
            if(intermat[i][j] < 0){
                cout << "Negative for " << allTracts[i]->getID() <<
                    " and " << allTracts[j]->getID() << endl;
                intermat[i][j] = max(myp[i],myp[j]);
            } else if(intermat[i][j] < 1e-5){
                intermat[i][j] = 0; //set to 0 so that they don't border
            }
            //cout << intermat[i][j] << endl;
        }
    }*/
    cout << "Done processing unions" << endl;
/*
    int sm = IDtoIref["1928646"];
    int top = IDtoIref["1928680"];
    int left = IDtoIref["1928388"];
    int rt = IDtoIref["1928582"];

    cout << myp[sm] << " " << intermat[sm][top] << " " <<
        intermat[sm][left] << " " << intermat[sm][rt] << endl;
    cout << myp[sm] << " " << myp[top] << " " << myp[left] << " "
        << myp[rt] << endl;

    for(i=0; i < NTRACT; i++){
        for(j=0; j < NTRACT; j++){
            if(bmat[i][j] >){
                allTracts[i]->addPerim(allTracts[j],intermat[i][j]);
            }
        }
    }
*/

    vector <Tract *> n;
    for(i=0; i < NTRACT; i++){
        for(j=0; j < NTRACT; j++){
            if(bmat[i][j]){
                n.push_back(allTracts[j]);
            }
        }
        allTracts[i]->setN(n);
        n.clear();
    }
    /*
    cout << myp[IDtoIref["754210"]] << endl;
    cout << myp[IDtoIref["759105"]] << endl;
    cout << intermat[IDtoIref["754210"]][IDtoIref["759105"]] << endl;
    cout << intermat[IDtoIref["578438"]][IDtoIref["593495"]] << endl;
    */
    //    " and " << calp->get_Length() << endl;
    //for(i=0; i < NDIST; i++){
    //    d[i]->getIsoPerim();
    //}
    District *dsm; //smallest district;
    District *dlg; // largest district;
    District *you,*me;
    vector<Fnode *> addingf;
    //Fnode *bestadd;
    County *iq;
    double pcscore,fcscore;
    double pcompactyou,pcompactme;
    double fcompactyou,fcompactme;
    double pvaryou,pvarme;
    double fvaryou,fvarme;
    double bestscore=-1e300;
    double tmpscore;
    // we do not need to consider my past compactness or my past
    // variance because all possible moves will consider that. Ignore.
    double varscore = 0;
    double pscore = -1e347;
    double curscore = -1e300;
    District *nextd;
    vector<Fnode *> adds;
    vector<Fnode *> removes;
    Fnode *bestadd;
    Fnode *bestremove;
    District *youtakeme;
    District *itakeyou;
    vector<District *> myborders;
    vector<Tract *> swappage;
    District *block;
    double prevscore,futscore;
    int count;
    for(count = 0; count < 500; count++){
        cout << "Iteration " << count+1 << endl;
        //do{
        pscore = curscore;
        // add to smallest District...
        bestadd = NULL;
        bestremove = NULL;
        bestscore = -1e300;

        nextd = nextD(d);
        adds = addingMoves(nextd);
        removes = reducingMoves(nextd);
        me = nextd;

        /*
        if(count < 200){
        myborders = me->whatBordersMe();
        block = myborders[randint(0,myborders.size()-1)];
        swappage = me->sharesBorder(block);
        // swap out, then swap in....
        prevscore = me->score() + block->score();
        for(i=0; i < swappage.size(); i++){
            moveTract(swappage[i],me);
        }
        futscore = me->score() + block->score();
        for(i=0; i < swappage.size(); i++){
            moveTract(swappage[i],block);
        }
        if((me->isContiguous() && block->isContiguous())){
            tmpscore = futscore - prevscore;
            if(tmpscore > 0){
                for(i=0; i < swappage.size(); i++){
                    moveTract(swappage[i],me);
                }
                cout << "Made massive swap!" << endl;
                continue;
            }
        }
        swappage = block->sharesBorder(block);
        for(i=0; i < swappage.size(); i++){
            moveTract(swappage[i],block);
        }
        futscore = me->score() + block->score();
        for(i=0; i < swappage.size(); i++){
            moveTract(swappage[i],me);
        }
        if((me->isContiguous() && block->isContiguous())){
            tmpscore = futscore - prevscore;
            if(tmpscore > 0){
                for(i=0; i < swappage.size(); i++){
                    moveTract(swappage[i],block);
                }
                cout << "Made massive swap!" << endl;
                continue;
            }
        }
        }*/
        // consider all adds
        for(i=0; i < adds.size(); i++){
            itakeyou = (adds[i]->getTract())->getDistrict();
            prevscore = itakeyou->score() + me->score();
            moveTract(adds[i]->getTract(),me);
            if(!itakeyou->isContiguous()){
                moveTract(adds[i]->getTract(),itakeyou);
                continue;
            }
            futscore = itakeyou->score() + me->score();
            tmpscore = futscore - prevscore;
            if(tmpscore > bestscore){
                bestscore = tmpscore;
                bestadd = adds[i];
                bestremove = NULL;
            }
            moveTract(adds[i]->getTract(),itakeyou);
        }

        // consider all removes
        for(i=0; i < removes.size(); i++){
            youtakeme = removes[i]->getDistrict();
            prevscore = youtakeme->score() + me->score();
            moveTract(removes[i]->getTract(),youtakeme);
            if(!me->isContiguous()){
                moveTract(removes[i]->getTract(),me);
                continue;
            }
            futscore = me->score() + youtakeme->score();
            tmpscore = futscore - prevscore;
            if(tmpscore > bestscore){
                bestscore = tmpscore;
                bestadd = NULL;
                bestremove = removes[i];
            }
            moveTract(removes[i]->getTract(),me);
        }

        // consider all swaps
        if(/*bestscore < 0 && randdub() < 0.9*/ true){
            for(i=0; i < removes.size(); i++){
                youtakeme =removes[i]->getDistrict();
                for(j=0; j < adds.size(); j++){
                    itakeyou = (adds[j]->getTract())->getDistrict();
                    if(youtakeme == itakeyou){
                        prevscore = youtakeme->score() + me->score();
                    } else {
                        prevscore = youtakeme->score() + me->score() +
                            itakeyou->score();
                    }
                    moveTract(removes[i]->getTract(),youtakeme);
                    moveTract(adds[j]->getTract(),me);
                    if(!itakeyou->isContiguous() ||
                            !me->isContiguous()){
                        moveTract(removes[i]->getTract(),me);
                        moveTract(adds[j]->getTract(),itakeyou);
                        continue;
                    }
                    if(youtakeme != itakeyou){
                        futscore = me->score() + youtakeme->score() +
                            itakeyou->score();
                    } else {
                        futscore = me->score() + youtakeme->score();
                    }
                    tmpscore = futscore - prevscore;
                    if(tmpscore > bestscore){
                        bestscore = tmpscore;
                        bestadd = adds[j];
                        bestremove = removes[i];
                    }
                    moveTract(removes[i]->getTract(),me);
                    moveTract(adds[j]->getTract(),itakeyou);
                }
            }
        }
        if(bestscore > 0){
            // make the moves, clear the stuff
            if(bestadd){
                moveTract(bestadd->getTract(),me);
            }
            if(bestremove){
                moveTract(bestremove->getTract(),bestremove->getDistrict());
            }
            if(bestadd && bestremove){
                cout << "Swap is the best move!" << endl;
            }
        }
        adds.clear();
        removes.clear();
        curscore = partTwoScore(d,allCounties);
        cout << "Current Score: " << curscore << endl;
        //} while(bestadd || bestremove);
        varscore = 0;
        for(i=0; i < NDIST; i++){
            varscore += d[i]->varScore();
        }
        //} while(varscore < -1);
}

int sumpump=0;
for(i=0; i < NDIST; i++){
    sumpump += d[i]->getPop();
    cout << "District␣" << i+1 << ":␣" << d[i]->getPop() << endl;
}

if(argv[2]){
ofstream ogil(argv[2]);
ogil << varscore << endl;
ogil << partTwoScore(d,allCounties) << endl;
list<Tract *> lst;
list<Tract *>::iterator liter;
for(i=0; i < NDIST; i++){
    ogil << "-1␣";
    lst = d[i]->getTractList();
    for(liter = lst.begin(); liter != lst.end(); liter++){
        ogil << (*liter)->getIndex() << "␣";
    }
    ogil << endl;
}

ogil.close();
}
cout << "Total␣population:␣" << sumpump << endl;
a = new Allocation(d);
plotAllocation(a, "parttwo_finish");
return 0;
}

//bool randnext = false;
District *nextD(District **d){

    District *smallest = d[randint(0,NDIST-1)];
    /*
    if(randnext){
        smallest = d[randint(0,NDIST-1)];
        randnext = false;

    } else {
        int i;
        smallest = d[0];
        double score = d[0]->score();
        double ts;
        for(i=1; i < NDIST; i++){
            ts = d[i]->score();
            if(ts < score){
                smallest = d[i];
                score = ts;
            }
        }
    randnext = true;
    }*/
    return smallest;
}


District *smallestD(District **d){
    int i;
    District *smallest = d[0];
    int smpop = d[0]->getPop();
    for(i=1; i < NDIST; i++){
        if(d[i]->getPop() < smpop){
            smpop = d[i]->getPop();
            smallest = d[i];
        }
    }
    return smallest;
}

District *largestD(District **d){
    int i;
    District *largest = d[0];
    int smpop = d[0]->getPop();
    for(i=1; i < NDIST; i++){
        if(d[i]->getPop() > smpop){
            smpop = d[i]->getPop();
            largest = d[i];
        }
    }
    return largest;
}

double partTwoScore(District **d,County **allCounties){
    int i;
    double compact=0,var=0,county=0,ncscore=0;
    for(i=0; i < NDIST; i++){
        compact += d[i]->compactScore();
        var += d[i]->varScore();
        county += d[i]->countyScore();
        ncscore += d[i]->newcountyScore();
    }

    /*
    for(i=0; i < NCOUNTY; i++){
        county += allCounties[i]->getValue();
    }*/
    cout << "Variance: " << var << " Compactness: " << compact <<
        " County: " << county << " New County Score: " << ncscore << endl;

    return var + compact + county + ncscore;
}

vector <Fnode *> reducingMoves(District *dis){
    list <Tract *> f = dis->getPerimeter();
    list <Tract *>::iterator liter;
    Fnode *tmp;
    vector <Fnode *> retval;
    int i;
    vector <District *> otherD;

    for(liter = f.begin(); liter != f.end(); liter++){
        otherD = (*liter)->getNColors();
        for(i=0; i < otherD.size(); i++){
            tmp = new Fnode(*liter,otherD[i]);
            retval.push_back(tmp);
        }
    }

    return retval;
}

vector <Fnode *> addingMoves(District *dis){
    list <Tract *> f = dis->getFrontier();
    list <Tract *>::iterator liter;
    Fnode *tmp;
    vector <Fnode *> retval;

    for(liter = f.begin(); liter != f.end(); liter++){
        tmp = new Fnode(*liter,dis);
        retval.push_back(tmp);
    }
    if(f.size() == 0){
        cout << "blank frontier" << endl;
    }
    if(retval.size() == 0){
        cout << "blank retval" << endl;
    }

    return retval;
}

vector <Fnode *> unionFrontier(District **d){
    int i;
    list<Tract *> f;
    list<Tract *>::iterator liter;
    Fnode *tmp;
    bool flag;
    vector <Fnode *> retval;
    for(i=0; i < NDIST; i++){
        f = d[i]->getFrontier();
        for(liter = f.begin(); liter != f.end(); liter++){
            if((*liter)->getDistrict() == BLANKDIST){
                tmp = new Fnode(*liter,d[i]);
                retval.push_back(tmp);
            }
        }
    }

    return retval;
}

// house cleaning to keep data structs in order
void moveTract(Tract *t, District *newd){
    District *oldd = t->getDistrict();
    if(oldd == newd){
        cerr << "Trying to change to already fixed district!" << endl;
        return;
    }
    list <Tract *> l = oldd->getTractList();
    l.remove(t);
    l = newd->getTractList();
    l.push_front(t);
    t->setDistrict(newd);
    oldd->removeFromDistrict(t);
    newd->addToDistrict(t);
}

void plotAllocation(Allocation *a,string fname){
    // plots an Allocation to a file

    const char *pszDriverName = "ESRI Shapefile";
    OGRSFDriver *poDriver;

    OGRRegisterAll();

    poDriver =
        OGRSFDriverRegistrar::GetRegistrar()->GetDriverByName(
                pszDriverName);
    if(!poDriver){
        cerr << "Could not initialize driver for writing!" << endl;
        return;
    }

    OGRDataSource *poDS;
    OGRLayer *layer;
    District **d = a->getDistricts();
    int i;
    string curname,lname;
    OGRFeature *tmpf;
    list <Tract *>tracts;
    list <Tract *>::iterator iter;

    for(i=0; i < NDIST; i++){
        tracts = d[i]->getTractList();
        curname = fname + inttostring(i) + ".shp";
        poDS = poDriver->CreateDataSource(fname.c_str(), NULL);
        if(!poDS){
            cerr << "Could not create output file!" << endl;
            return;
        }
        lname = "District " + inttostring(i+1);
        layer = poDS->CreateLayer(lname.c_str(), NULL, wkbUnknown,NULL);
        if(!layer){
            cerr << "Layer creation failed!" << endl;
            return;
        }

        for(iter = tracts.begin(); iter != tracts.end(); iter++){
            tmpf = new OGRFeature(layer->GetLayerDefn());
            tmpf->SetGeometry((*iter)->getGeo());
            if(layer->CreateFeature(tmpf) != OGRERR_NONE){
                cerr << "Could not create feature!" << endl;
                return;
            }
            OGRFeature::DestroyFeature(tmpf);
        }
        OGRDataSource::DestroyDataSource(poDS);
    }
}
```

```
#include "ogrsf_frmts.h"
#include <iostream>
#include <fstream>
#include <iomanip>
#include <string>
#include <map>
#include "Tract.h"
//#include "County.h"
#include "District.h"
#include "Allocation.h"
//#include "rng.h"
#include <sstream>
#include <cstdlib>
#include <ctime>
#include <vector>
#include "Fnode.h"
#include <algorithm>

const int NTRACT = 4907;
const int NDIST = 29;
const double AVGPEOPLE = 18976457./(float)NDIST;
const int NCOUNTY = 62;
//const int NLEVELS = 20;
District *BLANKDIST;
const bool PRINTHEU = false;

using namespace std;

void plotAllocation(Allocation *a,string fname);

string inttostring(const int i){
    ostringstream stream;
    stream << i;
    return stream.str();
}

int main(int argc, char *argv[]){
    srand((unsigned)time(NULL));

    OGRRegisterAll();

    OGRDataSource *myfile;

    myfile = OGRSFDriverRegistrar::Open("./polygons/", FALSE);
    if(myfile == NULL){
        cerr << "Can't open file" << endl;
        return 1;
    }
    cout << "Opened file appropriately!" << endl;
    cout << "File has " << myfile->GetLayerCount() << " layers" << endl;

    OGRLayer *layer = myfile->GetLayer(0);
    if(!layer){
        cerr << "Cannot open layer" << endl;
        return 1;
    }

    cout << "Layer has " << layer->GetFeatureCount() << " features" <<
        endl;
    int numtracts = layer->GetFeatureCount();
    int i,j;
    OGRFeature *feat;
    int populationindex;
    int totalpop = 0;
    map<string,int> IDtoIref;
    map<int,int> CkeytoRkey; // county key in file to our real keys.
    Tract *allTracts[NTRACT];
    bool **bmat = new bool*[NTRACT];
    double **distmat = new double*[NTRACT];

    Allocation *a;

    County **allCounties = new County*[NCOUNTY];
    for(i=0; i < NCOUNTY; i++){
        allCounties[i] = new County();
    }

    int cindex=-1;
    for(i=0; i < numtracts; i++){
        feat = layer->GetNextFeature();
        if(!feat){
            cerr << "Could not read feature, exiting!" << endl;
            return 1;
        }
        allTracts[i] = new Tract(feat,i);
        IDtoIref[allTracts[i]->getID()] = i;
        // Link to counties...
        if(CkeytoRkey.count(allTracts[i]->getCounty()) == 0){
            cindex++;
            CkeytoRkey[allTracts[i]->getCounty()] = cindex;
        }
        allCounties[CkeytoRkey[allTracts[i]->getCounty()]]->addToCounty(allTracts[i]);
        allTracts[i]->setCounty(allCounties[CkeytoRkey[allTracts[i]->getCounty()]]);

        delete feat;
        feat = NULL;
    }

    cout << "beginning to read border file..." << endl;
    ifstream bo;
    bo.open("border.txt");
    for(i=0; i < NTRACT; i++){
        bmat[i] = new bool[NTRACT];
        for(j=0; j < NTRACT; j++){
            bo >> bmat[i][j];
        }
```

```
     }
     bo.close();
     cout << "finished␣reading␣border␣file" << endl;
    District *d[NDIST+1]; // d[NDIST] = blank canvas....
    for(i=0; i < NDIST+1; i++){
            d[i] = new District();
    }

     vector <Tract *> n;
     for(i=0; i < NTRACT; i++){
         for(j=0; j < NTRACT; j++){
             if(bmat[i][j]){
                 n.push_back(allTracts[j]);
             }
         }
         allTracts[i]->setN(n);
         n.clear();
     }

     BLANKDIST = d[NDIST];
     // Read in file here....
     cout << "opening␣input␣file...." << endl;
     if(argc >= 2){
         list <Tract *>doolist;
         list <Tract *>::iterator liter;
         ifstream infile(argv[1]);
         if(!infile){
                 cerr << "Could␣not␣open␣" << argv[1] << endl;
                 return 1;
         }
         double upp;
         infile >> upp;
         cout << "Variance:␣" << upp << endl;
         infile >> upp;
         cout << "Score:␣" << upp << endl;
         int inp;
         for(i=-1; (i < NDIST) && !infile.eof(); i++){
             infile >> inp;
             while((inp != -1) && !infile.eof()){
                 allTracts[inp]->setDistrict(d[i]);
                 d[i]->addToDistrict(allTracts[inp]);
                 infile >> inp;
             }
         }

         infile.close();
     } else {
             cerr << "Must␣call␣an␣input␣file..." << endl;
             return 1;
     }

     for(i=0; i < NCOUNTY; i++){
             allCounties[i]->printCounty();
     }
     double varScore = d[0]->varScore();
     double cScore = d[0]->countyScore();
     double compact = d[0]->compactScore();
     double minpop = d[0]->getPop();
     double maxpop = d[0]->getPop();
     cout << "District␣" << 1 << "␣" << d[0]->getPop() << endl;
     for(i=1; i < NDIST; i++){
             //cScore += d[i]->countyScore();
             //compact += d[i]->compactScore();
             varScore += d[i]->varScore();
             if(d[i]->getPop() > maxpop){
                     maxpop = d[i]->getPop();
             }
             if(d[i]->getPop() < minpop){
                     minpop = d[i]->getPop();
             }
             cout << "District␣" << i+1 << "␣" << d[i]->getPop() << endl;
     }
     cout << "Variance:␣" << varScore << "␣Max:␣" << maxpop << "␣Min:␣"
             << minpop << endl;
     cout << "County:␣" << cScore << "␣Compact:␣" << compact << endl;

     a = new Allocation(d);
     plotAllocation(a,argv[2]);
     return 0;
    }

 void plotAllocation(Allocation *a,string fname){
     // plots an Allocation to a file

     const char *pszDriverName = "ESRI␣Shapefile";
     OGRSFDriver *poDriver;

     OGRRegisterAll();

     poDriver =
         OGRSFDriverRegistrar::GetRegistrar()->GetDriverByName(
                 pszDriverName);
     if(!poDriver){
         cerr << "Could␣not␣initialize␣driver␣for␣writing!" << endl;
         return;
     }

     OGRDataSource *poDS;
     OGRLayer *layer;
     District **d = a->getDistricts();
     int i;
     string curname,lname;
     OGRFeature *tmpf;
     list <Tract *>tracts;
     list <Tract *>::iterator iter;

     for(i=0; i < NDIST; i++){
         tracts = d[i]->getTractList();
         curname = fname + inttostring(i) + ".shp";
         poDS = poDriver->CreateDataSource(fname.c_str(), NULL);
         if(!poDS){
             cerr << "Could␣not␣create␣output␣file!" << endl;
             return;
         }
         lname = "District␣" + inttostring(i+1);
         layer = poDS->CreateLayer(lname.c_str(), NULL, wkbUnknown,NULL);
         if(!layer){
             cerr << "Layer␣creation␣failed!" << endl;
             return;
         }

         for(iter = tracts.begin(); iter != tracts.end(); iter++){
             tmpf = new OGRFeature(layer->GetLayerDefn());
             tmpf->SetGeometry((*iter)->getGeo());
             if(layer->CreateFeature(tmpf) != OGRERR_NONE){
                 cerr << "Could␣not␣create␣feature!" << endl;
                 return;
             }
             OGRFeature::DestroyFeature(tmpf);
         }
         OGRDataSource::DestroyDataSource(poDS);
     }
 }
```