Optimization Strategies for Political Redistricting

Blake Splitter
Clemson University, bsplitt@clemson.edu

Optimization Strategies for Political Redistricting

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Mathematics

---

by
Blake Splitter
August 2024

Abstract

Political redistricting has remained a hot-button issue in the United States for several decades. Every ten years, most states need to redraw their districts to account for changing populations. Sometimes, these district plans can be drawn with the malevolent intention of aiding one political party over another. This dissertation summarizes four distinct methods of drawing these districts using computer algorithms while keeping several objectives in mind. We test these approaches on the case study state of South Carolina, since it provides a sufficiently challenging problem for us to test various algorithms. We find that many of these approaches improve upon the current South Carolina district map, and many of the algorithms improve upon or remain competitive with other redistricting procedures available in the literature.

Dedication

This dissertation is dedicated to the people that fight for democracy in the U.S. and around the globe. It is easy to take freedom for granted, but the fragility of democracy demands constant attention and relentless defenders. For without the ideals of democracy and equality, this project would not even be possible.

Acknowledgments

I would like to acknowledge several people who have helped me in this seven-year journey as I worked to finish my Ph.D. In no particular order, thank you to:

- Clemson University's Palmetto Cluster support team; their supercomputer was invaluable as I conducted thousands of experiments, and could not have completed this dissertation without them;
- Dr. Gregory Herschlag for providing portions of his code for me to use;
- My advisor, Dr. Matthew Saltzman for his constant vigilance and guidance over the the entirety of my research journey at Clemson;
- My committee co-chair, Dr. Mary E. Kurz for her expertise in metaheuristics and regular advice as I progressed through the most challenging segment of this research;
- My other committee members, Dr. William Bridges and Dr. Neil Calkin for their feedback throughout the process;
- Amy Burton for her assistance with building portions of this code;
- Dr. Meredith Burr for her excellent supervisory skills as I learned to teach calculus as an instructor of record;
- Dr. Trevor Squires for his inspirational ideas during my Master's project;
- All of my friends and colleagues who kept me sane during my time as a graduate student;
- My family for their unwavering support during this journey, and particularly my brother Brandon for checking over my code on occasion;

- The late Dr. Kevin James for accepting me into Clemson;
- Anna-Marie Morra for her assistance near the beginning of this project.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the United States, 44 out of the 50 states have a sufficiently large population to send more than one representative to the House of Representatives. Each of these states must divide itself into two or more districts, each of which will elect a representative. The larger the population of the state, the more districts it has. The way in which these districts are drawn has a dramatic impact on who gets elected. Because populations change over time, the United States requires that every state redraws their districts every 10 years in response to the decennial census. This process is known as *redistricting*. With modern computing and enormous data sets on voting behavior, it has become very easy to draw the districts in such a way as to advantage one party over the other. This biased map drawing is known as *gerrymandering*, a term originally coined by the Boston newspaper *Weekly Messenger* in 1812, criticizing a district drawn by Governor Elbridge Gerry that vaguely looked like a salamander.

Although the Supreme Court of the United States has offered several decisions that place some constraints on how districts are drawn (see [2, 27, 39]), there is still ample opportunity to draw gerrymandered districts. In this dissertation, we argue that districts should be equally populated, politically fair, geographically compact, and county-preserving. This dissertation summarizes a variety of techniques that the author has developed over the past several years to solve the problem of dividing people into fair and equitable voting districts. We conduct our studies on congressional plans, but similar studies could be conducted on state houses or state senates.

When creating district plans, it is necessary to discretize the state into geographic units (GUs) that represent the smallest blocks of land that can be assigned to districts. In our work, we choose to use precincts as our geographic units since those are seldom split when assigning districts, though it is possible to consider finer discretizations such as census tracts or census blocks. Formally, we define a *plan* as an assignment of precincts to districts. In this dissertation, we choose to perform all of our algorithmic tests on the state of South Carolina, where Clemson University is located. In South Carolina, there were 2262 precincts in 2019, when the data was collected. Two of these precincts resided entirely inside other precincts, and were therefore merged into the enclosing precinct for our purposes.

District plans are constrained by the concept of contiguity. That is, every district must not have any disconnected pieces except in the case where landmasses can be considered adjacent if there is a body of water between them. Contiguity is a hard constraint that must not be broken at any point during any of the algorithms we propose.

1.1 Objectives Used

To assess the quality of a district plan, one must use a variety of objective measures. In the studies conducted in this dissertation, a total of six different objectives are considered. Each objective is defined so that it has an optimal value of zero and can only take on nonnegative values. Those objectives are:

1. **Population Deviation from Ideal,** PD: According to Supreme Court doctrine of "One Person, One Vote," established in cases Wesberry v. Sanders and Reynolds v. Sims, districts that elect representatives to state or federal legislatures need to be equally populated to the extent possible [27, 39]. This prevents the construction of districts with excess voting power due to their relatively small populations. In our algorithms, population deviation is measured using equation 1.1, where D is the set of districts, p_d is the population of the dth district, and $p_{\mathcal{I}}$ is ideal population of a district (i.e. the total population of the state divided by the number of districts). Because of the importance of population equinumerosity, this objective is considered in every approach taken in this dissertation, and in some cases is emphasized more strongly than other objectives.

$$PD = \sum_{d \in D} |p_d - p_{\mathcal{I}}| \tag{1.1}$$

2. **Polsby-Popper Compactness,** *PP*: District compactness is generally acknowledged as a necessary constraint to discourage gerrymandering. Broadly speaking, a district is compact if it is tightly packed in; it is not spread far. There are a variety of ways to measure a district's compactness, some that consider the geometry of the district, some that consider the distribution of people, and some that rely on the underlying graph. We direct the reader to [12] for a more comprehensive view of several compactness measures. In this dissertation, we utilize Polsby-Popper compactness, which is a geometric measure that calculates the ratio of a district's area to the area of a circle with the same perimeter as the district, though other compactness measures work as well. We opt to use this compactness measure because (1) it is easy to compute, (2) it is used in at least one state's (Arizona's) independent redistricting commission, and (3) it passes the eye-test for what most people consider "compact" [16]. Normally, Polsby-Popper scores have a range between 0 and 1, with 1 corresponding to an ideally compact district (i.e, a circle). However, because we intend to minimize each objective, and wish to define its optimal value as zero, we either calculate the *shifted* Polsby-Popper score, detailed in equation (1.2); or the *inverse* Polsby-Popper score, detailed in equation (1.3). In both equations, D is the set of districts, A_d is the area of the dth district, and ρ_d is the perimeter of the dth district.

$$PP_s = \frac{1}{|D|} \sum_{d \in D} \left(1 - \frac{4\pi A_d}{\rho_d^2}\right) \tag{1.2}$$

$$PP_i = \frac{1}{|D|} \sum_{d \in D} \left(\frac{\rho_d^2}{4\pi A_d}\right) - 1 \tag{1.3}$$

3. **Efficiency Gap,** *EG*: Originally developed by Stephanopoulos and McGhee, the efficiency gap is an objective that seeks to distinguish fair plans from unfair plans [32]. Essentially, this objective counts the number of 'wasted' votes cast for the two main parties and determines if the difference between those two totals signifies an unfair plan. To predict this measure before elections, we often use past voter behavior to predict future voting behavior. The efficiency gap is calculated by subtracting the wasted votes of party B from the wasted votes of party A and then dividing this difference by the total votes. Wasted votes are (1) all votes for the losing side and (2) all votes beyond the win threshold for the winning side. We calculate the efficiency gap for each district and add those values together to get the statewide efficiency

gap. We desire for the absolute efficiency gap to be minimized, since a large efficiency gap in magnitude represents a partisan advantage. The calculation for absolute efficiency gap can be found in equation (1.4), where ω_{Ad} represents the wasted votes for party A in the dth district, ω_{Bd} represents the wasted votes for party B in the dth district, and t represents the total vote count of the state.

$$EG = \left| \sum_{d \in D} \left(\frac{\omega_{Ad} - \omega_{Bd}}{t} \right) \right| \tag{1.4}$$

4. **Median-Mean fairness score,** MM: The median-mean fairness score is a measure of how 'fair' a plan is politically. The premise underlying this objective is that district plans that do not reflect the mean voter distribution in a state are undesirable. For instance, if 55% of a state's voters voted for one political party, then we would want about 55% of the seats in the legislature to represent that political party. To represent this mathematically, consider the set $\mathcal{S}$ of party A's vote share percentages in each district and calculate both the mean and median of this set. The median-mean score is calculated by subtracting the mean from the median. Positive values for this objective indicate an advantage for party A, while negative objective values indicate a disadvantage for party A. Values near zero are desirable and demonstrate that the districts approximately reflect the will of the voters. McDonald and Best explain finer details of the median-mean calculation and how it could demonstrate gerrymanders [21]. In our calculations, it is desirable to minimize the absolute value of the median-mean score because we want the advantage of the parties to be as small as possible. This calculation is found in equation (1.5).

$$MM = |\text{median}(\mathcal{S}) - \text{mean}(\mathcal{S})| \tag{1.5}$$

5. **County Splits,** CS: A county split occurs when a county is split between two or more districts. Because counties often have a sense of shared community (from shared laws and government), it is desirable to avoid splitting counties into multiple districts. The number of county splits is found by subtracting the minimum of the number of counties and districts from the number of county-district intersections, as detailed in equation (1.6), where CDI represents the number of county-district-intersections, C represents the set of counties, and D is the set of districts.

$$CS = CDI - \min(|C|, |D|) \tag{1.6}$$

6. **Excess Geographic Units,** *EGU*: Excess geographic units represent the number of the GUs in a county that are not part of the largest district in that county (as measured by number of GUs). Minimizing excess geographic units is a conduit to minimizing county splits, so optimizing both objectives better aims to reduce county splits. The number of excess geographic units is found using equation (1.7), where GU_c is the number of GUs in county c and GU_{cd} is the number of GUs that are in both county c and district d.

$$EGU = \sum_{c \in C} (GU_c - \max_{d \in D}(GU_{cd})), \tag{1.7}$$

The projects that follow utilize a subset of these objectives. There are several other objectives that we do not consider in this dissertation, including (a) adherence to the Voting Rights Act, (b) racial makeup of districts, (c) preservation of communities of interest, (d) similarity to previous plans, and (e) incumbent location. We have assessed that these objectives are too difficult to mathematically model, lack reliable data, or would not improve the district plans if considered. Furthermore, every additional objective potentially decreases the quality of each proposed plan as we must sacrifice existing objectives to account for new ones.

1.2 Dissertation Outline

This dissertation is divided as follows. In Chapter 2, we explore the numerous redistricting strategies that other authors have employed so far. In Chapter 3, we explain how the redistricting problem can be written as a mixed-integer-program. To follow, we discuss a simulated annealing approach for maximizing compactness in Chapter 4. Then we expand on the simulated annealing concept in Chapter 5, which brings other objectives into a multi-objective framework. Finally, we discuss a genetic algorithm approach in Chapter 6, and make concluding remarks in Chapter 7.

Chapter 2

Literature Review

In the United States, every state's Congressional map must be redrawn every decade to account for changing populations. Gerrymandering is the process of creating voting districts that unfairly advantage one group of people over another. The problem of creating voting districts that fairly represent their populations is one of the most well-studied problems in political science, and the impulse to remove human bias from redistricting seems a natural one. In fact, the desire to automate the redistricting process dates back to at least the early 1960s when Vickrey writes that, "[T]he [redistricting] process should be completely mechanical, so that once set up, there is no room at all for human choice" [37]. Since then, there have been many attempts to utilize computers in an effort to automate district drawing.

It is important to distinguish our work as a project meant to *create* district plans; many other authors study the important, but very different problem of assessing districts plans after they have been proposed. Early work on this front was done by quantifying the compactness of legislative districts, arguing that noncompact districts may be evidence of gerrymandering. In view of this, many authors developed "compactness tests" to pinpoint gerrymandering [10, 23, 26]. Other authors have sought to uncover gerrymandering with political fairness tests. These tests use previous voting data as an estimate for future voting patterns. Based on these predictions, authors develop metrics to determine how 'fair' a district plan is. Tests such as median-mean [21], efficiency gap [32], and partisan asymmetry [38] all fall into this category of work. Finally, many important works seek to determine how unfair a district plan is by comparing it to a host of reasonable alternative plans. Ideally, we would enumerate every possible districting plan and compare the proposal to the universe of options, but the number of legal districting plans is incomprehensibly large, making any enumerative strategy effectively impossible. The seminal work by Duchin uses Markov chain outlier analysis to determine that a Pennsylvania redistricting plan was likely chosen explicitly to provide partisan advantage to Republicans [11]

In contrast to studying gerrymandering ex post facto, our work focuses primarily on eliminating the gerrymander before it can emerge. Many works before us have sought to create district plans using computer automation in some way or another. Several authors utilize Voronoi approaches (a technique designed to divide many data points into clusters) to create compact districts of relatively equal populations [e.g. 19, 25, 33]. A variety of metaheuristic techniques have been attempted, including simulated annealing, tabu search, old bachelor acceptance, genetic algorithms, and more. Most of these approaches optimize a single weighted-sum objective function consisting of one or more metrics meant to indicate a good districting plan. See for example [5], [20], and [28].

Other authors have sought direct integer programming approaches. This work starts in 1965 with Hess et al., where they worked to optimize compactness via moment of inertia geometric calculations [14]. This work modeled a hypothetical IP to solve this problem, without actually solving it. In 1983, Birge formulated a quadratic program to minimize the number of county splits [4]. This work was later eclipsed by Shahmizad and Buchanan in 2023, when they found the optimal county split value for every state [30]. In 2021, Gurnee et al. developed a dynamic column-generation heuristic for solving IPs meant to optimize efficiency gap [13]. Heuristics were necessary to fully solve these IPs. Finally, Belloti et al. attempted to optimize compactness using counties as geographic units in 2023 [3].

More recently, there have been several attempts to utilize multicriteria optimization in redistricting. These experiments yield a set of viable solutions that seek to optimize two or more objective functions. The outcome set of districting plans estimates a Pareto frontier. Vanneschi et al. uses a multiobjective approach based on a genetic algorithm to create an approximate Pareto front for compactness and population equality [36]. Kim uses a spanning tree approach to create artificial districts on a $n \times n$ grid of squares, optimizing both population equality and compactness [17]. In 2022, Swamy performs a case study on Wisconsin's congressional districts, optimizing political fairness metrics against compactness using an integer programming approach [35].

The work in this dissertation follows previous approaches by two different authors. Chapter 5 follows the work done by Rincón-García [29]. Much of our work in this chapter is based on their initial invention, which is a multiobjective simulated annealing (MOSA) algorithm that optimizes compactness and population equinumerosity. We have expanded upon this idea in this paper with MOSA2, by adding many additional objectives and modifying the way in which perturbations are performed.

Chapter 6 follows the work done by Vanneschi et al. in [36]. This work utilized the Non-dominated Sorting Genetic Algorithm-II (NSGA-II) to optimize compactness and population deviation in a variety of states. We adapt the approach to incorporate more objectives and implement a novel clustering approach to better fit the theme of genetic algorithms.

Chapter 3

Mixed-Integer Programming Approach

The first attempt at optimizing political redistricting is the very direct approach of mixed-integer programming. This technique hopes to optimize every individual objective subject to population and contiguity constraints. The plans produced through this approach may provide a baseline for multiobjective approaches, introduced in later chapters.

3.1 Formulations

3.1.1 Feasible Plans

In order to find a feasible districting plan, we mimic a formulation from Shirabe [31]. We insist that districts are contiguous and that they satisfy population equinumerosity constraints.

The decision variables are as follows:

- $x_{ij} = 1 \iff$ geographic unit i is assigned to district j.
- $w_{ij} = 1 \iff$ geographic unit i is assigned to be the hub for district j.
- f_{uvj} is the "flow" on edge (u, v) in district j. This quantity is positive if and only if both endpoints u and v are in district j and is equal to 0 otherwise. These variables are the ones used for enforcing contiguity.

The sets used are:

- V: the set of vertices in the underlying graph G. This set is normally indexed by i, u, or v.
- E: the set of edges in the underlying graph G. This set is normally indexed by (u, v), where $u, v \in V$ and (u, v) is an edge in the graph G.
- D: the set of districts. This set is normally indexed by j.

Other data used includes:

- $\mathcal{L}$: the lower bound on population for all districts. This can be user-defined.
- $\mathcal{U}$: the upper bound on population for all districts. This can be user-defined.
- $\mathcal{P}_i$: the population in geographic unit i.

$$
\begin{array}{llll}
\boxed{\text{FEAS}} & & & \\
\min & 0 & & \\
\text{S.t.} & \displaystyle\sum_{j \in D} x_{ij} = 1 & \forall i \in V & (3.1) \\
& \mathcal{L} \leq \displaystyle\sum_{i \in V} \mathcal{P}_i x_{ij} \leq \mathcal{U} & \forall j \in D & (3.2) \\
& \displaystyle\sum_{i \in V} w_{ij} = 1 & \forall j \in D & (3.3) \\
& \displaystyle\sum_{v:(u,v) \in E} f_{uvj} - \sum_{v:(u,v) \in E} f_{vuj} \geq x_{uj} - |V| w_{uj} & \forall u \in V, j \in D & (3.4) \\
& \displaystyle\sum_{v:(v,u) \in E} f_{uvj} \leq (|V| - 1) x_{uj} & \forall u \in V, j \in D & (3.5) \\
& x_{ij} \in \mathbb{B} & \forall i, j \in D & (3.6) \\
& w_{ij} \in \mathbb{B} & \forall i \in V, j \in D & (3.7) \\
& f_{uvj} \in \mathbb{R}^{+} & \forall (u, v) \in E, j \in D & (3.8)
\end{array}
$$

The constraints can be thought of as follows:

- Constraint (3.1) defines that every geographic unit i is assigned to exactly one district j.
- Constraint (3.2) defines the population constraints of each district j.

- Constraint (3.3) defines that each district gets exactly one hub.
- Constraints (3.4) and (3.5) define the flow from a district hub $v \in V$ to other geographic units $u \in V$ in district j.
- Constraints (3.6), (3.7), and (3.8) define the domains for the variables.

We can add a variety of objectives and appropriate defining constraints to create MIPs that find optimality with respect to those objectives.

3.1.2 Population Deviation

We often consider population equinumerosity of the districts as a constraint, rather than an objective, but it can be advantageous to find a plan that minimizes population deviation from ideal. To do so, we define the decision variable d_j to be the absolute population deviation from ideal for district j. We also define $\mathcal{I}$ to be the ideal population of a district. Note that $\mathcal{I} = \frac{1}{|D|}\sum_{i \in V} \mathcal{P}_i$. Then the MIP that minimizes the sum of population deviations over all districts is:

$$\boxed{\text{POP}}$$

$$\begin{aligned}
\min \quad & \sum_{j \in D} d_j && && (3.9)\\
\text{S.t.} \quad & d_j \geq \sum_{i \in V} \mathcal{P}_i x_{ij} - \mathcal{I} && \forall j \in D && (3.10)\\
& d_j \geq \mathcal{I} - \sum_{i \in V} \mathcal{P}_i x_{ij} && \forall j \in D && (3.11)\\
& d_j \in \mathbb{Z}^+ && \forall j \in D && (3.12)\\
& (3.1), (3.3) - (3.8)
\end{aligned}$$

Here, constraints (3.10) and (3.11) ensure that we are maximizing the sums of the absolute differences between population and ideal. Constraint (3.2) is excluded, because minimizing population deviation renders lower and upper bounds on population useless.

3.1.3 Compactness

In this chapter, we opt to minimize the inverse Polsby-Popper score. We can do that by defining the following variables:

- z_j = inverse Polsby-Popper score of district j.
- a_j = area of district j.
- ρ_j = perimeter of district j.
- $y_{uvj} = 1 \iff$ edge (u, v) is a *cut edge* (i.e., an edge that separates two districts) and geographic unit u is in district j.

Other data includes:

- $\mathcal{E}_i$ is the boundary edge length for geographic unit i. In other words, if geographic unit i is on the state's boundary, $\mathcal{E}_i$ represents the length of that boundary. If geographic unit i is not on the state's boundary, $\mathcal{E}_i = 0$.
- $\mathcal{S}_{uv}$ is the boundary length of edge $e := (u, v)$ between geographic units u and v.
- $\mathcal{A}_i$ is the area of geographic unit i.

Utilizing the formulation from Belotti et al., we establish the quadratically-constrained MIP for compactness with the following minimization problem [3].

$$\boxed{\text{COMP}}$$

$$\begin{aligned}
\text{min} \quad & \frac{1}{|D|}\sum_{j \in D} z_j & & (3.13)\\
\text{S.t.} \quad & 4\pi A_j z_j \geq \rho_j^2 & \forall j \in D \quad & (3.14)\\
& \rho_j = \sum_{i \in V} \mathcal{E}_i x_{ij} + \sum_{(u,v) \in E} \mathcal{S}_{uv} y_{uvj} & \forall j \in D \quad & (3.15)\\
& A_j = \sum_{i \in V} \mathcal{A}_i x_{ij} & \forall j \in D \quad & (3.16)\\
& x_{uj} - x_{vj} \leq y_{uvj} & \forall (u,v) \in E, j \in D \quad & (3.17)\\
& x_{vj} - x_{uj} \leq y_{uvj} & \forall (u,v) \in E, j \in D \quad & (3.18)\\
& z_j \geq 0 & \forall j \in D \quad & (3.19)\\
& y_{uvj} \in \mathbb{B} & \forall (u,v) \in E, j \in D \quad & (3.20)\\
& (3.1) - (3.8) & &
\end{aligned}$$

Objective 3.13 minimizes the average Polsby-Popper scores amongst all districts j. The constraints can be explained as follows:

- Constraint (3.14) defines the inverse Polsby-Popper score.
- Constraint (3.15) defines the perimeter of district j.
- Constraint (3.16) defines the area of district j.
- Constraints (3.17) defines the cut edges y_{uvj} as described above.

3.1.4 County Splits

County splits occur when two or more districts occupy a single county. If n districts occupy a single county, then that county has $n - 1$ county splits. It is desirable to avoid splitting counties into multiple districts. To optimize this, we define the decision variable:

- $h_{cj} = 1 \iff$ county c contains any part of district j.

We use the set C for the set of counties in the state.

We also introduce new data:

- $\mathcal{C}_{ic} = 1 \iff$ geographic unit i is in county c. In South Carolina, each geographic unit is entirely within the boundary of a single county.

$$\boxed{\text{CS}}$$

$$\min \quad \sum_{c \in C} \sum_{j \in D} h_{cj} \tag{3.21}$$

$$\text{S.t.} \quad \frac{1}{|V|} \sum_{i \in V} \mathcal{C}_{ic} x_{ij} \leq h_{cj} \qquad \forall c \in C, j \in D \tag{3.22}$$

$$h_{cj} \in \mathbb{B} \qquad \forall c \in C, j \in D \tag{3.23}$$

$$(3.1) - (3.8)$$

The objective (3.21) minimizes the total number of county splits by summing over both $c \in C$ and $j \in D$. Constraint (3.22) defines h_{cj} as defined above. Essentially, the quantity $\frac{1}{|V|} \sum_{i \in V} \mathcal{C}_{ic} x_{ij}$ equals 0 if no geographic units in county c are assigned to district j and is ≤ 1 otherwise.

3.1.5 Efficiency Gap

To minimize the absolute efficiency gap, let

- r_j = number of votes for red party in district j.
- b_j = number of votes for blue party in district j.
- t_j = winning threshold for district j. $(= \frac{b_j+r_j}{2} + 0.5)$

Normally, efficiency gap is calculated by dividing by the total number of votes. In this study, we instead omit that division, since minimizing $\frac{\text{wasted blue}-\text{wasted red}}{\text{total}}$ is the same as minimizing (wasted blue)$-$(wasted red). Then for a given district j, the efficiency gap in that district would be

$$\begin{aligned} EG_j &= (\text{wasted blue votes}) - (\text{wasted red votes}) \\ &= \begin{cases} (b_j - t_j) - r_j, & \text{if } b_j \geq r_j \\ b_j - (r_j - t_j), & \text{if } b_j < r_j \end{cases} \\ &= \begin{cases} \left(b_j - \left(\frac{b_j+r_j}{2} + 0.5\right)\right) - r_j, & \text{if } b_j \geq r_j \\ b_j - \left(r_j - \left(\frac{b_j+r_j}{2} + 0.5\right)\right), & \text{if } b_j < r_j \end{cases} \\ &= \begin{cases} \frac{b_j}{2} - \frac{3r_j}{2} - 0.5, & \text{if } b_j \geq r_j \\ \frac{3b_j}{2} - \frac{r_j}{2} + 0.5, & \text{if } b_j < r_j \end{cases} \end{aligned} \tag{3.24}$$

Then, minimizing (3.24) is the same as minimizing

$$\begin{cases} b_j - 3r_j - 1, & \text{if } b_j \geq r_j \\ 3b_j - r_j + 1, & \text{if } b_j < r_j \end{cases}$$

Therefore, we seek to minimize

$$\min \quad \left| \sum_j \Big[(b_j - 3r_j - 1) \cdot \mathbb{I}(b_j \geq r_j) + (3b_j - r_j + 1) \cdot \mathbb{I}(r_j > b_j) \Big] \right| \tag{3.25}$$

where $\mathbb{I}$ is the indicator function. Absolute value bars are placed outside the sum because we want the efficiency gap to be as small (in magnitude) as possible. The bias towards blue or red is irrelevant for the calculation.

This can be enforced with the following MIP:

$$\boxed{\text{EG}}$$

$$\min \quad z \tag{3.26}$$

$$\text{S.t.} \quad b_j = \sum_i \mathcal{B}_i x_{ij} \qquad \forall j \in D \tag{3.27}$$

$$r_j = \sum_i \mathcal{R}_i x_{ij} \qquad \forall j \in D \tag{3.28}$$

$$z \geq \sum_j \left[(b_j - 3r_j - 1)\mathbb{I}_j + (3b_j - r_j + 1)\mathbb{I}_j\right] \tag{3.29}$$

$$z \geq -\sum_j \left[(b_j - 3r_j - 1)\mathbb{I}_j + (3b_j - r_j + 1)\mathbb{I}_j\right] \tag{3.30}$$

$$\frac{b_j - r_j}{\mathcal{N}} \leq \mathbb{I}_j \leq 1 + \frac{b_j - r_j}{\mathcal{N}} \qquad \forall j \in D \tag{3.31}$$

$$b_j, r_j \in \mathbb{Z}^+ \qquad \forall j \in D \tag{3.32}$$

$$\mathbb{I}_j \in \mathbb{B} \qquad \forall j \in D \tag{3.33}$$

$$(3.1) - (3.8)$$

where

- $\mathcal{B}_i$ is the number of blue votes in the ith GU
- $\mathcal{R}_i$ is the number of red votes in the ith GU
- $\mathcal{N}$ is the total number of votes in South Carolina.

Then,

- $\mathbb{I}_j = \begin{cases} 1, & \text{if } b_j \geq r_j \\ 0, & \text{if } b_j < r_j \end{cases}$

Constraints (3.27) and (3.28) define blue votes and red votes in district j. Constraints (3.29) and (3.30) are used to minimize the absolute value of the objective function in (3.25). Constraints (3.31) define the indicator function $\mathbb{I}_j$.

3.1.6 Median-Mean

In our calculations, it is desirable to minimize the absolute value of the median-mean score because we want the advantage of the parties to be as small as possible. We define the following decision variables:

- $\mathcal{B}_i$ represents the blue votes in GU i.
- $\mathcal{R}_i$ represents the red votes in GU i.
- z represents the absolute median-mean calculation.
- b_j represents the blue votes in district j.
- r_j represents the red votes in district j.
- β_j represents the blue-share-of-votes $\left(\beta_j = \frac{b_j}{b_j + r_j}\right)$.
- s_k represents the sorted β values. In particular, s_k is the kth smallest β value.
- σ represents a $|D| \times |D|$ permutation matrix.
- μ represents the median value amongst all β values.

To optimize the absolute median-mean fairness score, we utilize MIP $\boxed{\text{MM1}}$:

FORMULATION 1

$$\boxed{\text{MM1}}$$

$$
\begin{array}{llll}
\min & z & & (3.34)\\
\text{S.t.} & b_j = \sum_{i \in V} \mathcal{B}_i x_{ij} & \forall j \in D & (3.35)\\
& r_j = \sum_{i \in V} \mathcal{R}_i x_{ij} & \forall j \in D & (3.36)\\
& (b_j + r_j)\beta_j = b_j & \forall j \in D & (3.37)\\
& s_k = \sum_{j \in D} \sigma_{jk}\beta_j & \forall k \in D & (3.38)\\
& s_{k+1} \geq s_k & \forall k \in D' & (3.39)\\
& \sum_{j \in D} \sigma_{jk} = 1 & \forall k \in D & (3.40)\\
& \sum_{k \in D} \sigma_{jk} = 1 & \forall j \in D & (3.41)\\
& \mu = s_{(|D|-1)/2} & \text{if } |D| \text{ is odd} & (3.42)\\
& \mu = \frac{1}{2}(s_{(|D|-2)/2} + s_{|D|/2}) & \text{if } |D| \text{ is even} & (3.43)\\
& z \geq \mu - \frac{1}{|D|}\sum_{j \in D} \beta_j & & (3.44)\\
& z \geq \frac{1}{|D|}\sum_{j \in D} \beta_j - \mu & & (3.45)\\
& b_j, r_j \in \mathbb{Z}^+ & \forall j \in D & (3.46)\\
& \mu, z \geq 0 & & (3.47)\\
& \beta_j, s_j \in \mathbb{R}^+ & \forall j \in D & (3.48)\\
& \sigma_{jk} \in \mathbb{B} & \forall j, k \in D & (3.49)\\
& (3.1) - (3.8) & &
\end{array}
$$

where $D' = D$ except that D' excludes the last element of D, when D is sorted.

Here, constraints (3.35) and (3.36) define the blue and red votes for each district j. Constraint (3.37) defines the blue-share-of-votes variables β_j for each district j. Constraint (3.38) permutes the variables β_j onto s_k. Then (3.39) orders the s_k variables. Constraints (3.40) and (3.41) define the permutation matrix σ. Constraints (3.42) and (3.43) define the median variable μ. Constraints (3.44) and (3.45) define z to be absolute value of the median-mean quantity. Constraints (3.46) - (3.49) define the various variable domains.

If we know that there are an odd number of districts, we can use an alternate formulation, where rather than sorting the β values, we instead force μ to be the median value via big-M constraints.

FORMULATION 2: Let $H = (|D| - 1)/2$ represent half of the number of districts, rounded down. Let m_j be a binary variable, where $m_j = 1 \iff \mu \geq \beta_j$. Similarly, let n_j be a binary variable where $n_j = 1 \iff \mu \leq \beta_j$. Also define M to be a sufficiently large number. Then the second formulation to optimize the absolute median-mean is:

$\boxed{\text{MM2}}$

$$\min \quad z \tag{3.50}$$

$$\text{S.t.} \quad b_j = \sum_{i \in V} \mathcal{B}_i x_{ij} \qquad \forall j \in D \tag{3.51}$$

$$r_j = \sum_{i \in V} \mathcal{R}_i x_{ij} \qquad \forall j \in D \tag{3.52}$$

$$(b_j + r_j)\beta_j = b_j \qquad \forall j \in D \tag{3.53}$$

$$\sum_{j \in D} m_j = H + 1 \tag{3.54}$$

$$\sum_{j \in D} n_j = H + 1 \tag{3.55}$$

$$\beta_j - \mu \leq M(1 - m_j) \quad \forall j \in D \tag{3.56}$$

$$\mu - \beta_j \leq M(1 - n_j) \quad \forall j \in D \tag{3.57}$$

$$z \geq \mu - \frac{1}{|D|} \sum_{j \in D} \beta_j \tag{3.58}$$

$$z \geq \frac{1}{|D|} \sum_{j \in D} \beta_j - \mu \tag{3.59}$$

$$b_j, r_j \in \mathbb{Z}^+ \qquad \forall j \in D \tag{3.60}$$

$$\mu, z \geq 0 \tag{3.61}$$

$$\beta_j \in \mathbb{R}^+ \qquad \forall j \in D \tag{3.62}$$

$$m_j, n_j \in \mathbb{B} \qquad \forall j \in D \tag{3.63}$$

$$(3.1) - (3.8)$$

Constraints (3.56) and (3.57) force μ to take on the value of the median β value. Once again, this assumes an odd number of districts, else the problem would be infeasible.

3.2 Methods

All models above are developed by the author of this dissertation, with the exception of $\boxed{\text{FEAS}}$ and $\boxed{\text{COMP}}$. These models are all formulated in Python using Gurobi's Python interface Gurobipy, where the "NonConvex" parameter was set at 2 to deal with the various quadratic constraints. The code used to create these models can be found in Appendix A: MIP Codes. Each model had a maximum time limit of 71 hours and was run on Clemson University's Palmetto Cluster. Each MIP utilized one node in the Palmetto Cluster, 10 CPUs, 120GB of memory, and a Fourteen Data Rate (FDR) interconnection. The number of districts was set at seven and two experiments were run with population deviations from ideal set at $\pm 5\%$ and $\pm 1\%$. In other words, $(\mathcal{L},\mathcal{U})_{5\%} = \left(\frac{0.95}{|D|}\sum_{i\in V}\mathcal{P}_i, \frac{1.05}{|D|}\sum_{i\in V}\mathcal{P}_i\right)$ and $(\mathcal{L},\mathcal{U})_{1\%} = \left(\frac{0.99}{|D|}\sum_{i\in V}\mathcal{P}_i, \frac{1.01}{|D|}\sum_{i\in V}\mathcal{P}_i\right)$ in constraint (3.2). To improve performance, a feasible plan with a population deviation of 483 was fed into the model to supply starting values for the assignment decision variables x_{ij}. This feasible plan was created by running the MIP that optimizes population deviation, with an objective cutoff of ≤ 500. That starting plan is pictured in Figure 3.1.

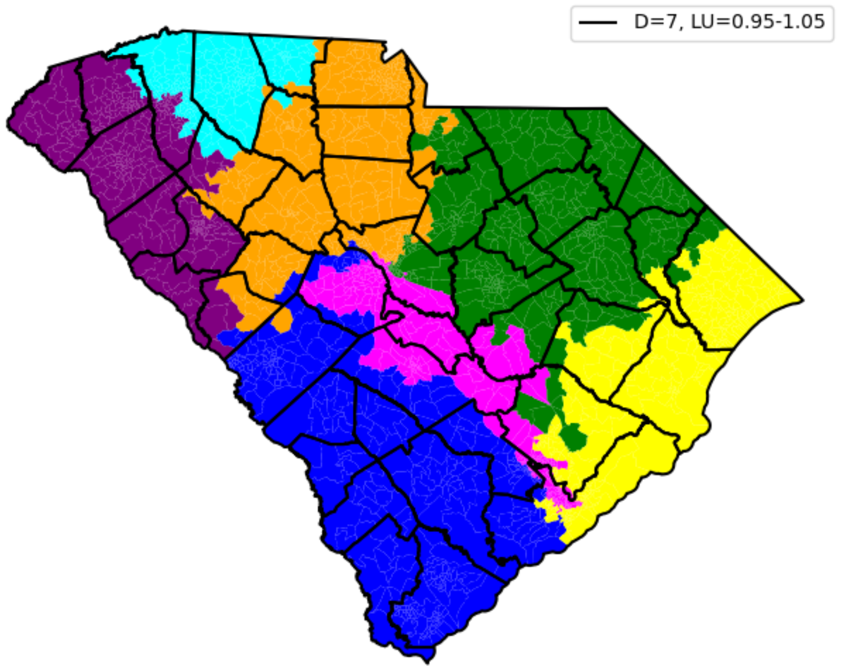


Figure 3.1: Starting Plan

Some constraints that dealt with small decision variables were multiplied by a large constant so that Gurobi wouldn't accept suboptimal solutions due to insufficiently small tolerances. Constraints (3.56)–(3.59) were all multiplied by 1000 since the subtractions in those constraints can yield small results that Gurobi may otherwise round to 0.

3.3 Results

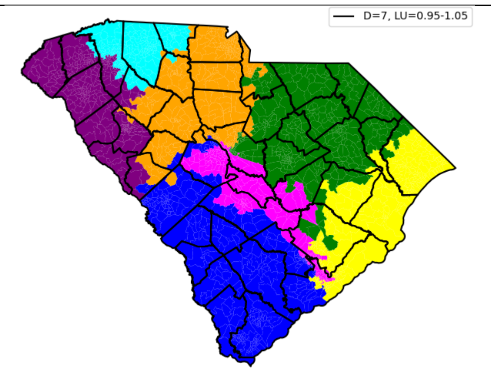
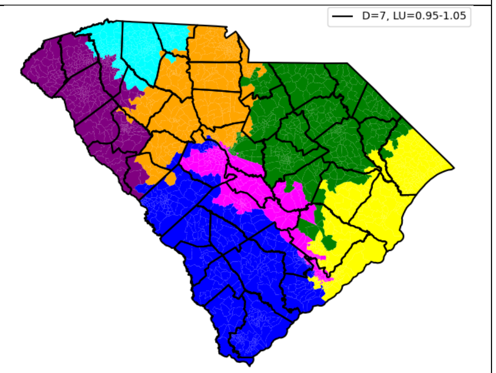
After running the experiments described in the Methods section, we report the best objective values found in Tables 3.1 and 3.2. Any results with run times that have an asterisk did not solve to optimality; those runs all hit the time limit of 71 hours (255600 seconds).

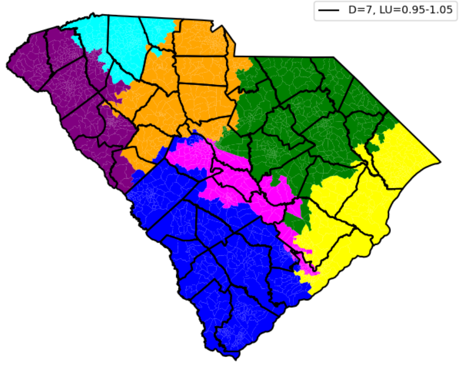
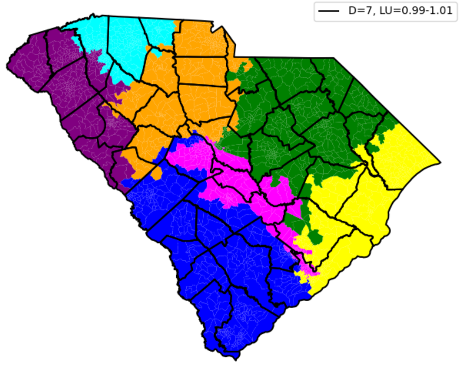
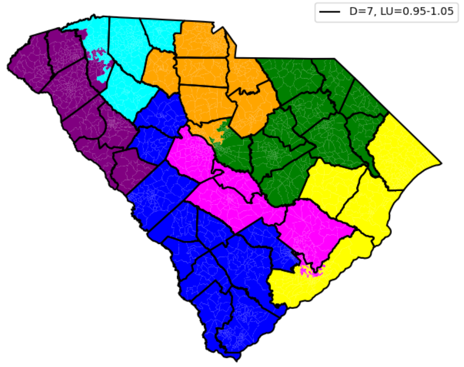
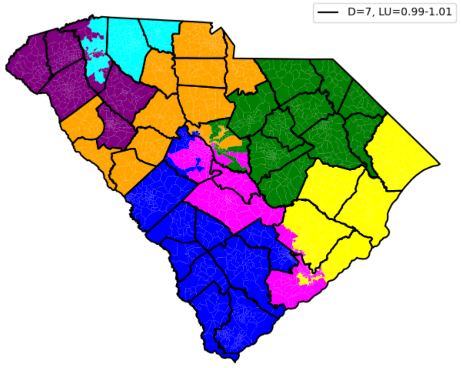
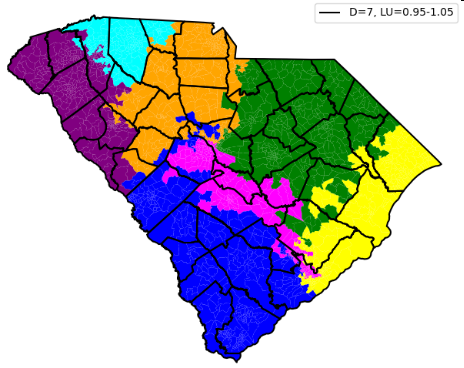
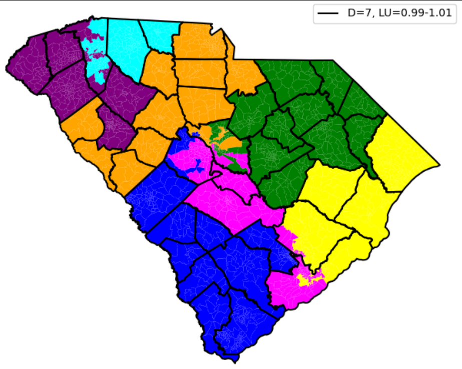
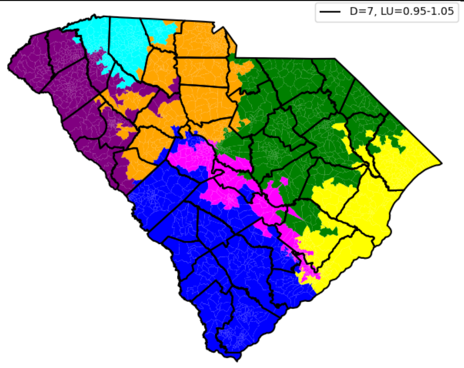
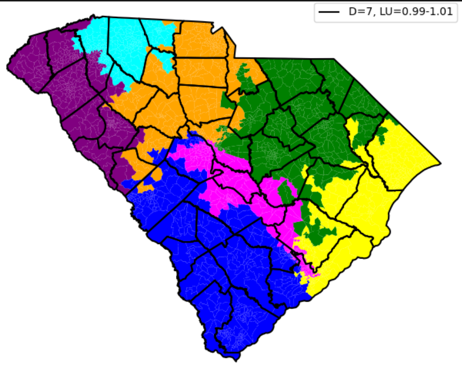
Objective	Runtime (s)	Initial Obj	Best Obj Found	Pop Dev
Population Deviation	255600*	483	9	9
Compactness	255600*	6.852	5.899	59569
County Splits	255600*	30	3	186917
Efficiency Gap	255600*	0.1127	0.0945	197263
Median-Mean (form 1)	45320	0.00451	0.00000	136395
Median-Mean (form 2)	592	0.00451	0.00000	123231

Table 3.1: MIP Results for ±5% Population Deviation

Objective	Runtime (s)	Initial Obj	Best Obj Found	Pop Dev
Compactness	255600*	6.852	6.852	483
County Splits	255600*	30	7	28923
Efficiency Gap	255600*	0.1127	0.1058	37353
Median-Mean (form 1)	4577	0.00451	0.00000	33769
Median-Mean (form 2)	976	0.00451	0.00000	35347

Table 3.2: MIP Results for ±1% Population Deviation

Objective	±5% plan	±1% plan
Population Deviation		

Compactness		
County Splits		
Efficiency Gap		
Median-Mean (form 1)		

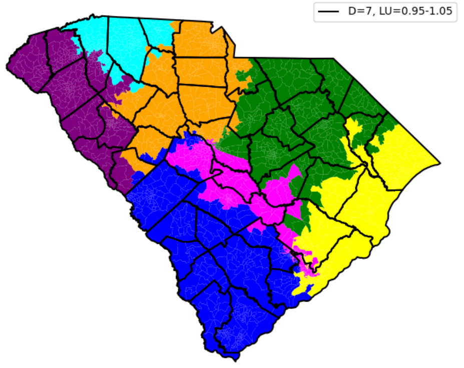
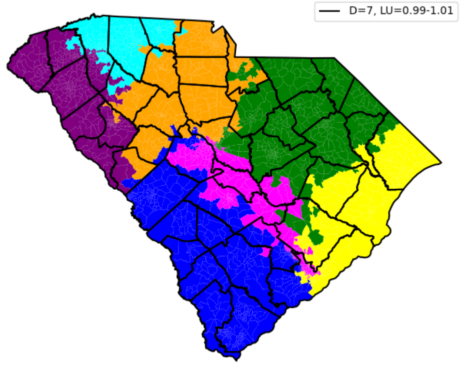
Median-Mean (form 2)		

Table 3.3: Optimal plans from MIP

3.4 Conclusion

After observing the results of the various MIPs, a few glaring concerns emerge. First, many of the experiments were unable to solve in the 71-hour time limit. This is not entirely unexpected, as each formulation has hundreds of thousands of variables and tens of thousands of constraints. Secondly, even with 71 hours of processing time, many of the formulations fail to make much progress. This is most apparent with $\boxed{\text{COMP}}$ with $\pm 1\%$ population deviation constraints, which made no progress over the course of the solve. The authors of the paper from which $\boxed{\text{COMP}}$ is defined only used the formulation on county graphs, so perhaps poor performance on the precinct graph should be expected. Third, the pictorial evidence seems to suggest that many of the resultant plans bear a striking resemblance to the starting plan seen in Figure 3.1. This suggests that the MIP is incapable of solving these models to optimality without serious modifications to the approach.

While the MIP experimentation may seem disheartening, this study does yield some important results. Firstly, it's important to note that these results can be used as a measuring stick for the heuristics in the following chapters. We should probably expect that the MIP performs better in each individual objective compared to the multiobjective approaches. The best objectives found in tables 3.1 and 3.2 can be used as ideal points for the sake of multiobjective optimization.

Chapter 4

Optimizing Compactness Using Simulated Annealing

4.1 Introduction and Overview

Judging by the results in Chapter 3, we can see that sometimes, mathematical programs cannot always be solved to optimality in any reasonable time frame. The number of variables and objectives exceeds the capabilities of exact solution strategies.

Thus, we turn our attention to heuristics. This chapter reports on a metaheuristic approach to optimize district compactness for the congressional districts of South Carolina. In theory, this approach could be utilized to optimize any objective, but compactness is easy to visualize. A secondary objective considered in this chapter is county splits. Population deviation is set as a constraint.

The specific metaheuristic approach used is simulated annealing. Simulated annealing is a metaheuristic technique that seeks to maximize or minimize an objective function that may have many local optima. Simulated Annealing is particularly useful in combinatorial problems where large numbers of configurations are possible. For example, if one were to randomly assign each of the 2260 precincts in South Carolina to one of the seven congressional districts (with no regard for contiguity), the number of possible arrangements would be 5.92×10^{19}, far too large a number to consider every possibility.

Broadly, simulated annealing works as follows:

- Define a state space Ω, where a given state $c \in \Omega$ *neighbors* another state $n \in \Omega$ if a single perturbation of c can create n.
- Define a real-valued function E ("E" for "energy") that needs to be minimized. The energy of state c is $E(c)$.
- Start at some feasible state c.
- Establish a "high" starting temperature T.
- From the current feasible state, randomly consider a neighbor n. Let $\Delta E := E(n) - E(c)$.
  - If $\Delta E < 0$ (i.e., transitioning to the neighbor would improve the objective by decreasing it), then accept the transition and move to the neighbor. Redefine the current state: $c := n$.
  - If $\Delta E > 0$ (i.e., it is a worsening move), accept the transition with probability $\frac{1}{\exp(\Delta E)/T}$. If accepted, set $c := n$.
- Gradually, over the course of the algorithm, reduce T.
- Stop when a local optimum is reached.

The key idea is that at the beginning of the algorithm, when T is large (relative to $\exp(\Delta E)$), transitions from one state to another are done essentially at random. Then, near the end of the algorithm, transitions that improve (decrease) the energy of the system are done almost exclusively.

Simulated annealing has been used in many applications, including the traveling salesman problem, circuit board design, school timetabling, molecule structure design, and more [1, 18, 34]. Critically, D'Amico et al. used simulated annealing to produce compact police districts [6]. This study was particularly inspirational for solving our problem.

4.2 Our Simulated Annealing Algorithm

We start by defining a graph G, where each geographic unit (i.e., each precinct) is assigned a node and an edge is placed between nodes if there is a nonzero length boundary between them. Any

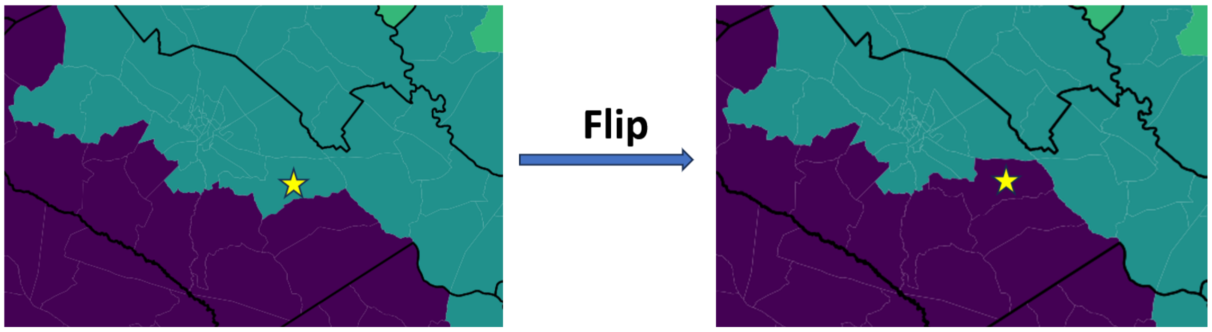


Figure 4.1: Demonstrating a Flip

enclave geographic units (GUs) are merged into their enclosing GUs. Any island GUs are declared to be adjacent to the GU that is geographically closest to them.

The state space Ω is the set of all feasible plans that assign every geographic unit to a district $d \in D$, where D is the set of all districts. A plan is considered feasible if each district is contiguous. Contiguity in this context requires that for every pair of geographic units g_a and g_b assigned to the district d, there is a path $g_a, g_1, g_2, \ldots, g_b$ where each node in the path is also in district d.

In order to move from one plan c to a neighboring plan n, we perform a *perturbation* on plan c. This perturbation could be many things, but the simplest option (and the one that is used in this study) is to flip a single GU on a district border from one district to another. Figure 4.1 demonstrates an example of a flip. The starred GU changes from the lighter-colored district to the darker-colored district.

For each flip, we need to ensure that the district losing a precinct is still contiguous. If such a flip creates a discontiguous district, we reject the perturbation. However, checking the contiguity of all GUs in the shrinking district is computationally expensive, so we instead only check whether the subgraph of all GUs assigned to the shrinking district that are also within distance 2 (measured by number of edges between nodes) of the transitioning GU are contiguous. This procedure is far less computationally expensive, but it occasionally rejects flips that actually are legal. It never accepts an illegal flip, however.

A secondary and less strict constraint is population deviation. Similar to its definition in subsection 3.1.2, population deviation is the sum of each district's deviation from the ideal population of the district. Constitutionally, this value must be as small as possible (generally below 1% in many cases). However, in order to successfully explore the state space, we allow this deviation to grow arbitrarily large before later requiring perturbations that improve population deviation.

The procedure used in this study differed from traditional simulated annealing (TSA) in several important ways. Those differences are documented in the following subsections.

4.2.1 Weighting Candidate Flips

Rather than picking a single neighboring plan and either accepting or rejecting that transition, we select several candidate neighbors and assign a weight to each one based on their energy. Then, using a random weighted selection, we transition to that neighbor. This also means that potential transitions that reduce the energy of the system may not be implemented. The weight for the jth candidate precinct flip is

$$W_j = \frac{S_j^p \cdot A}{1 + \exp\left(\frac{\Delta E_j}{T}\right)}, \tag{4.1}$$

where

- ΔE_j is the "change in energy" from the current state to the proposed state. This is discussed further in subsection 4.2.3.
- T is the current temperature of the system.
- S_j is a population scaling factor. If the flip improves population balance between the districts, then $S_j > 1$. If the flip worsens the population balance, then $S_j \in (0, 1)$. If the flip keeps both modified districts in the desired population window, then $S_j = 1$. More precisely,

$$S_j = \begin{cases} 1 + 0.1(q_\ell + q_e), & \text{if } (q_\ell + q_e) > 0 \\ 1, & \text{if } q_\ell + q_e = 0 \\ 0.9^{-(q_\ell + q_e)}, & \text{if } q_\ell + q_e < 0 \end{cases} \tag{4.2}$$

  where q_ℓ and q_e are population differences (in percentage points) from the acceptable population window for the leaving and entering districts, respectively. To be precise, if the acceptable population window for a district is $[L, U]$, the ideal population is I, and the population of the leaving and entering districts are pop_ℓ and pop_e, then:

$$q_\ell = \begin{cases} \frac{pop_\ell - U}{0.01I}(> 0), & \text{if } pop_\ell > U \\ \frac{pop_\ell - L}{0.01I}(< 0), & \text{if } pop_\ell < L \\ 0, & \text{if } L < pop_\ell < U \end{cases} \tag{4.3}$$

$$q_e = \begin{cases} \frac{pop_e - U}{0.01I}(< 0), & \text{if } pop_e < U \\ \frac{pop_e - L}{0.01I}(> 0), & \text{if } pop_e > L \\ 0, & \text{if } L < pop_e < U \end{cases} \tag{4.4}$$

- The exponent p increases the impact of the population scaling factor S_j. Population power p increases over the course of the algorithm. Conceptually, this is done to encourage the code to care more about population balance as we advance throughout the algorithm. In particular, the value of p is documented in Table 4.1.

- A is a binary variable that determines whether the flip maintains district contiguity. $A = 1$ if contiguity is maintained, and 0 otherwise.

Percentage of SA Steps Taken	Population Power p
$[0, 20)$	1
$[20, 40)$	2
$[40, 60)$	4
$[60, 80)$	8
$[80, 90)$	16
$[90, 95)$	32
$[95, 97.5)$	64
$[97.5, 100]$	128

Table 4.1: Population power p as a function of percent of simulated annealing (SA) steps taken.

It is worth noting that the denominator in Equation 4.1 differs from TSA by adding 1. This is done since candidate flips with $\Delta E_j < 0$ are not automatically accepted. Therefore, without the $+1$, the denominator would approach 0 for some values of ΔE_j and W_j would explode to $+\infty$. To prevent this outcome, we ensure that the denominator is always at least 1.

4.2.2 Objectives

Traditional simulated annealing optimizes a single objective. Our modification attempts to simultaneously optimize two. Those objectives are district compactness and county splits. We measure compactness using inverse Polsby-Popper scores, detailed in equation 1.3 from Chapter 1, and also listed below for reference. Recall that D is the set of districts, r_d is the perimeter of the dth district, and A_d is the area of the dth district.

$$PP_i = \frac{1}{|D|} \sum_{d \in D} \left(\frac{r_d^2}{4\pi A_d} \right) - 1$$

The secondary objective is county splits. As mentioned in Chapter 1, a county split occurs when one county is occupied by two or more districts. It is desirable to avoid this if possible.

4.2.3 Changes to ΔE

In TSA, an energy is calculated for the current state and the neighboring (proposed) state. Then $\Delta E = E_n - E_c$, where E_c is the current energy and E_n is the energy for the neighboring state. If $\Delta E < 0$, then the transition to the neighboring state is made; if $\Delta E > 0$, then the flip occurs with a certain probability as discussed above. In this algorithm however, we instead perform a standalone calculation for ΔE; that is, we opt not to calculate the energy for each individual state, and instead calculate a 'score' for the proposed transition. That ΔE score is based on the objectives defined in subsection 4.2.2. In order to minimize both objectives, we invent the following ΔE function:

$$\Delta E = B(\Delta x, \Delta C)^I \cdot (|\Delta C| + 1)^L \cdot \text{sign}(\Delta C) \tag{4.5}$$

The variables are best explained through example. In Figure 4.2, Sumter county is split between three districts of various colors.[1] Suppose the proposed flip moves the starred precinct P from the blue district to the white district. Then:

- $\Delta x := x_\ell - x_e$, where x_ℓ is the number of precincts in both P's county and the leaving district. x_e is the number of precincts in both P's county and the entering district. In the example, $x_\ell = 5$ and $x_e = 52$, so $\Delta x = -47$.

[1]These districts are present outside Sumter county as well, but we focus in on a single county.

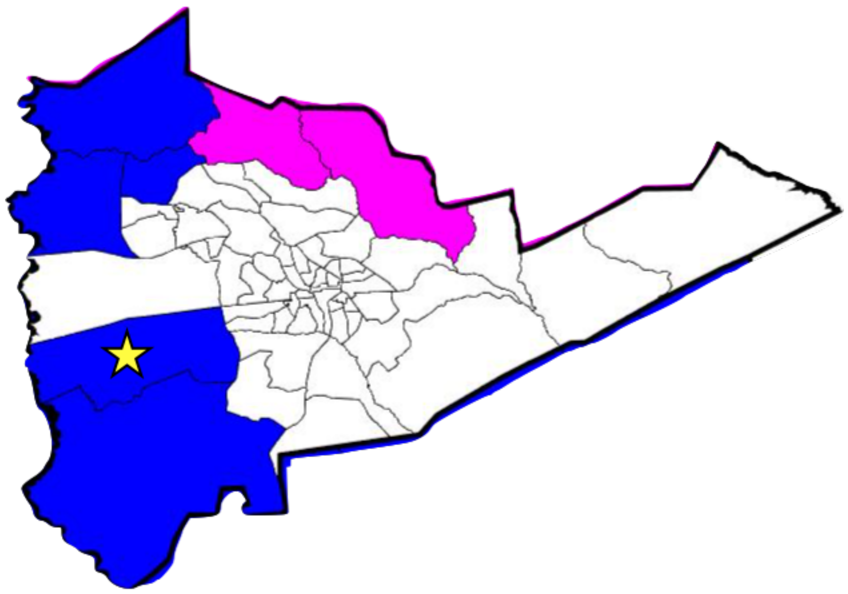


Figure 4.2: Here, Sumter County has parts of three districts in it.

- $\Delta C := C_n - C_c$, where C_n is the inverse Polsby-Popper compactness score after the proposed flip, and C_c is the current inverse Polsby-Popper compactness score.[2]
- I is a static, user-specified binary indicator; $I = 1$ if we care about county boundaries, and $I = 0$ otherwise.
- L is a static, user-specified positive value. The larger L is, the more that we emphasize compactness compared to county splits.
- $B(\cdot)$ is a *bolstering* function. It either amplifies or diminishes the value of ΔE. In particular,

$$B(\Delta x, \Delta C) = \begin{cases} 1 + (\Delta x)^{2/3}, & \text{if } \operatorname{sign}(\Delta x) = \operatorname{sign}(\Delta C) \\ \dfrac{1}{1 + (\Delta x)^{2/3}}, & \text{otherwise} \end{cases} \tag{4.6}$$

Notice that $B(\Delta x, \Delta C) > 0$ in all cases. The idea behind this function is that if Δx and ΔC share the same sign, then they are harmonious measures, and $B(\Delta x, \Delta C)$ is a "bolstering" function; it falls in the range $[1, \infty)$, thereby increasing the magnitude of ΔE. Conversely, if Δx and ΔC do not share the same sign, then they are conflicting measures. For example,

[2]We use inverse Polsby-Popper values here to better distinguish compact plans from non-compact plans.

Δx \ ΔC	-0.25	-0.125	0	0.125	0.25
-25	-11.94	-10.74	0	0.12	0.13
-5	-4.91	-4.41	0	0.29	0.32
0	-1.25	-1.13	0	1.13	1.25
5	-0.32	-0.29	0	4.41	4.91
25	-0.13	-0.12	0	10.74	11.94

Table 4.2: Table of Sample ΔE Values ($I = L = 1$)

the proposed precinct flip may improve compactness (negative ΔC) but worsen county splits (positive Δx). In this case, $B(\Delta x, \Delta C)$ resides in $(0, 1]$, thereby decreasing the magnitude of ΔE.

Overall, this formulation of ΔE achieves the following things:

- $\Delta E < 0$ if compactness would improve with the flip.
- $\Delta E > 0$ if compactness would worsen with the flip.
- $\Delta E \in (-\infty, -1)$ if both compactness and county splits would improve with the flip.
- $\Delta E \precapprox (-1, 0)$ if compactness would improve but county splits would worsen with the flip.[3]
- $\Delta E \precapprox (0, 1)$ if compactness would worsen but county splits would improve with the flip.
- $\Delta E \in (1, \infty)$ if both compactness and county splits would worsen with the flip.

These results are illustrated in Figure 4.3. We further justify this function by showing some sample Δc and Δx values in Table 4.2.

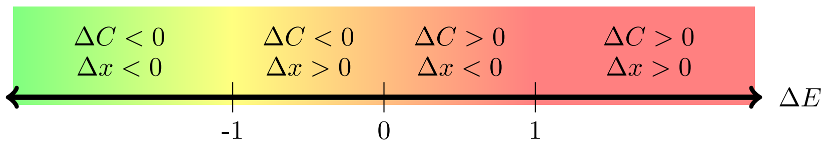

$\Delta C < 0$ $\Delta x < 0$
$\Delta C < 0$ $\Delta x > 0$
$\Delta C > 0$ $\Delta x < 0$
$\Delta C > 0$ $\Delta x > 0$
-1
0
1
ΔE

Figure 4.3: The values of ΔE based on the values of ΔC and Δx

4.2.4 Cooling Schedules

Simulated Annealing algorithms are equipped with *cooling schedules*. These describe the method by which temperature decreases. Much work has been done comparing various cooling

[3]The symbol $\precapprox$ means "approximately within." These values can occasionally escape the intervals.

schedules. See for example [24]. In this work, we present three cooling schedules:

A. Hill climbing: For the first 50% of iterations, $T = 100$, and for the last 50% of iterations, $T = 0.1$. This technique is tested as a baseline to see if gradual cooling (used by schemes B and C) has any beneficial effect.

B. Geometric cooling: $T_{k+1} = \alpha T_k$. The value α is the geometric ratio and is user-specified as $\alpha \in [0.95, 0.999]$.

C. Heating then geometric cooling: We keep the system at its initial temperature until the compactness score starts to level out, or we reach the halfway point of the user-specified maximum iterations. Then we start geometric cooling with $\alpha \in [0.9, 0.999]$.

In schedules B and C, the Markov chain length is 1000 precinct flips at each temperature. The code terminates after 1 million iterations. In all schedules, the initial temperature is set to be $T = 100$.

4.2.5 Algorithm Pseudocode

The algorithm's pseudocode is presented in Algorithm 1.

4.3 Experimental Design

To formally conduct experiments to find the plan with the best compactness and the one that respects county boundaries the most, we ran the algorithm several times at different input settings. Specifically, there are five input variables that we manipulated:

1. L: The exponent on the compactness measure. The larger this value, the more the code values compact districts. A default value of 1 works for this parameter, but it could be as arbitrarily large as we want it to be.

2. I: The indicator variable that determines whether we desire to preserve county boundaries. I is set to 1 if county splits are to be considered and 0 otherwise.

3. n: The number of boundary precincts sampled in each iteration.

4. Cooling Schedule (CS): The method by which we decrease temperature.

Algorithm 1 Main Code

INPUT:

L: The exponent on compactness scores
I: The indicator variable that determines whether we desire to preserve county boundaries
n: The number of boundary precincts sampled in each iteration
Cooling Schedule: the method by which we decrease temperature
α: The geometric ratio used when using cooling schedule B or C
Itmax: The maximum number of iterations to perform

```
1:  for $i = 1 : \text{Itmax}$ do
2:      Determine temperature $T$ (a decreasing function of iteration number)
3:      Determine population power $p$ (an increasing function of iteration number)
4:      flag= 0
5:      while flag= 0 do
6:          Select $n$ precincts on district boundaries
7:          for $j = 1 : n$ do
8:              Determine how the $j$th precinct flip would affect district compactness
9:              Calculate $\Delta E_j$ for the $j$th precinct flip
10:             Calculate population scale variable $S_j$ for the $j$th precinct flip
11:             Calculate the binary adjacency check variable $A$
12:             Assign a weight $W_j$ for the $j$th precinct flip candidate, using the formula
                $W_j = \left(S_j^p \cdot A\right) / \left(1 + \exp\left(\Delta E_j / T\right)\right)$
13:         end for
14:         if Any weight $\neq 0$ (indicating that at least one flip does not break contiguity) then
15:             flag= 1
16:             Using a weighted selection, choose a precinct to flip over district lines
17:         end if
18:     end while
19:     Update district perimeter, area, compactness score, and population;
20: end for
```

5. α: The geometric ratio used when using cooling schedule B or C. The variable α determines how quickly the geometric decrease for temperature occurs.

To conduct our experiments, we choose a small sample of test points for our input values. The possible test points are as follows:

1. $L \in \{1, 10, 50\}$
2. $I \in \{0, 1\}$
3. $n \in \{10, 30, 50\}$
4. Cooling Schedule $\in \{A, B, C\}$
5. $\alpha \in \{0.985, 0.99\}$

Because there are $3 \times 2 \times 3 \times 3 \times 2 = 108$ different possible experimental designs with these input test values, we opt to only test a subset of ten of these experiments. We used D-optimality to choose particular experiments which we hypothesize will produce differing results. In particular, we choose the subset of ten designs so that the collection of experiments makes the input variables as independent as possible. We eventually desire to know which input values cause the biggest change in our compactness score; we claim that our subset of ten experiments can accomplish that. For a more in-depth discussion of D-optimality, see The National Institute of Standards and Technology's description [22]. The input values for the ten experiments conducted can be found in Table 4.3.

Experiment Number	L	I	n	Cooling Schedule	α
1	10	0	10	C	0.99
2	1	0	50	C	0.985
3	50	1	30	B	0.985
4	10	1	30	C	0.985
5	1	1	10	B	0.985
6	50	0	50	A	–
7	50	1	10	A	–
8	50	1	10	C	0.99
9	10	1	50	A	–
10	10	0	50	B	0.99

Table 4.3: Input Data for each Experiment

In each of these ten experiments, we run the algorithm back-to-back ten times, so the final plan created on run one is the first plan used for run two. Therefore, for each set of input variables, the algorithm could run for as many as 10 million iterations. The purpose of doing this back-to-back so many times is to ensure that each run has a unique starting point and will produce different results each time. Further, the goal of this study is to validate the claim that compactness can be improved from the current congressional plan. We argue that this experimental design accomplishes that goal.

One other concern of ours is to ensure the algorithm does not unnecessarily spend time doing more iterations once a local optimum is reached. To prevent unnecessary iterations, we allow the algorithm to "skip ahead" if two conditions are satisfied:

1. The current compactness score is less than half of the maximum compactness score reached, and
2. The slope of the best fit line for the last 2000 iterations is in the interval $[-0.000025, 0.000025]$. This ensures that the average change in the last 2000 scores is between -0.05 and 0.05. This indicates that a local optimum has likely been reached.

These conditions are checked every 1000 iterations. When the algorithm skips ahead, it skips to the next 10% benchmark. So if the algorithm is currently at iteration 546,000, and the two conditions above are satisfied, it skips to iteration 600,000 (the 60% benchmark) and adopts the appropriate temperature T and population power p for that iteration value. This test is not applied in the final 10% of iterations.

4.4 Results

The 2019 congressional district plan for South Carolina can be found in Figure 4.4. At the time that this study was conducted, this was the current congressional plan. In this district drawing, the inverse Polsby-Popper score is 7.7096 and the number of county splits is 12. The experiments we run seek to improve these objectives. We use this 2019 congressional plan as our starting plan for each of the experiments performed. The input values for the ten experiments are found in Table 4.3.

The results for each of the ten experiments are summarized in Table 4.4. In each experiment, the algorithm is run ten times back-to-back. This code was compiled in Matlab and run on Clemson University's supercomputer using ten cores, an Intel Xeon chip, and 120 GB of memory. The specific code can be found in Appendix B: Simulated Annealing Codes. Each statistic in this

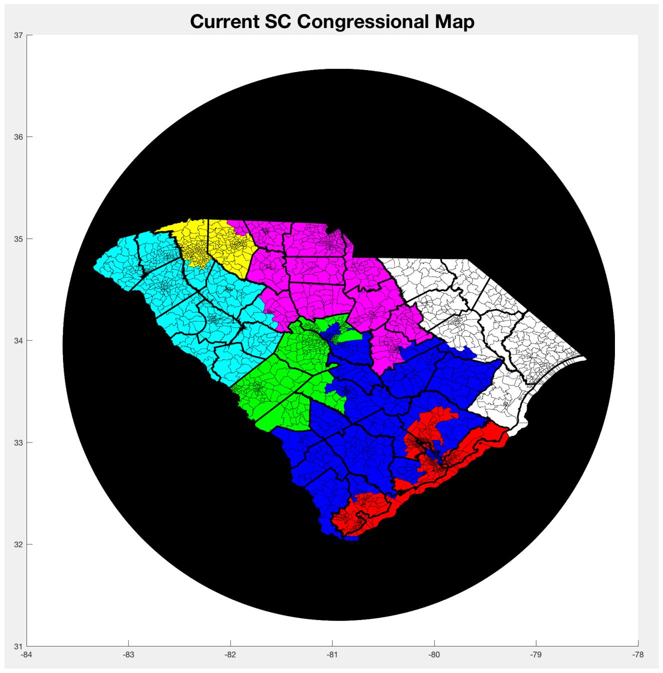


Figure 4.4: The 2019 South Carolina Congressional Plan

table summarizes those ten runs. For instance, "Best PP_i" displays the best inverse Polsby-Popper score found amongst the ten runs. "Mean PP_i" tabulates the average of the *best* inverse Polsby-Popper scores found in each of the ten runs. A similar idea holds in columns 4 and 5, where we analyze county splits. Further, every statistic in columns 2–5 originated out of a plan with district populations within $\pm 1\%$ of the ideal district population of 690,660 people. Column 6 identifies the worst instance of the inverse Polsby-Popper measure found. This column corresponds to a plan that does not necessarily meet population constraints. Column 7 averages the number of iterations needed in each run of the algorithm.

Next, we highlight some of the starkest results. Out of the tens of millions of iterations, the most compact plan is produced in experiment 3. The inverse Polsby-Popper score is 1.65 and this plan can be seen in Figure 4.5. For comparison, the plan which best emphasizes county boundaries and had the fewest county splits (only 8) is found in experiment 4. This plan can be seen in Figure 4.6. Both plans met the population constraint. The plan with the absolute worst compactness score across all experiments is found in experiment 2. This plan had an astronomical inverse Polsby-Popper score of 152.76 and can be seen in Figure 4.7. This is an example of a plan that is reached in the heating phase of the algorithm, and we do not guarantee that such a plan satisfies population constraints.

Exp. No.	Best (min) PP_i	Mean PP_i	Min CS	Mean CS	Worst (max) PP_i	Mean Iterations
1	2.36	2.71	21	23.7	118.28	546,300
2	2.04	2.46	19	20.8	152.76	582,800
3	1.65	2.12	10	12	10.53	248,700
4	2.57	4.01	8	11	85.24	345,300
5	4.35	4.87	14	16.7	146.00	495,500
6	3.03	3.54	24	29	24.19	613,300
7	2.64	2.76	10	11	14.89	625,400
8	2.64	2.81	10	12.4	12.71	312,100
9	3.08	4.91	9	11.9	96.57	630,900
10	2.31	3.01	21	24.5	109.78	523,500

Table 4.4: Summary Data per Experiment

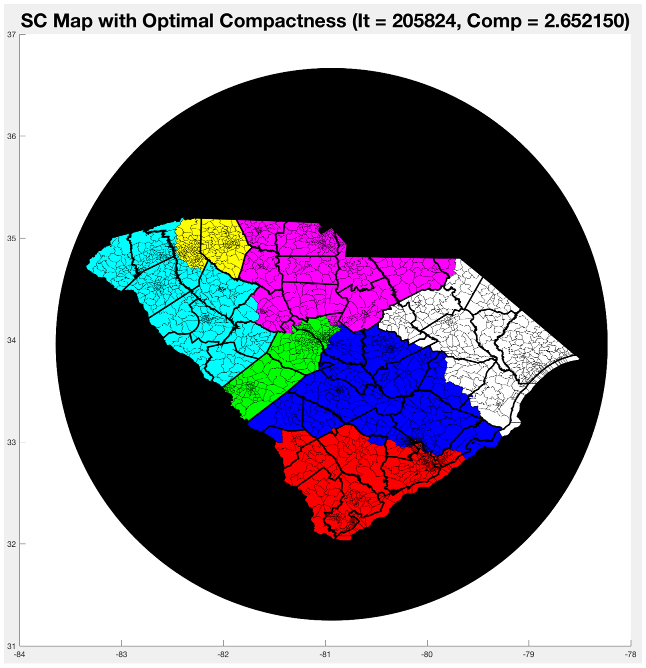

SC Map with Optimal Compactness (It = 205824, Comp = 2.652150)
37
36
35
34
33
32
31
-84
-83
-82
-81
-80
-79
-78

Figure 4.5: The Most Compact Plan ($PP_i = 1.65$), Experiment 3

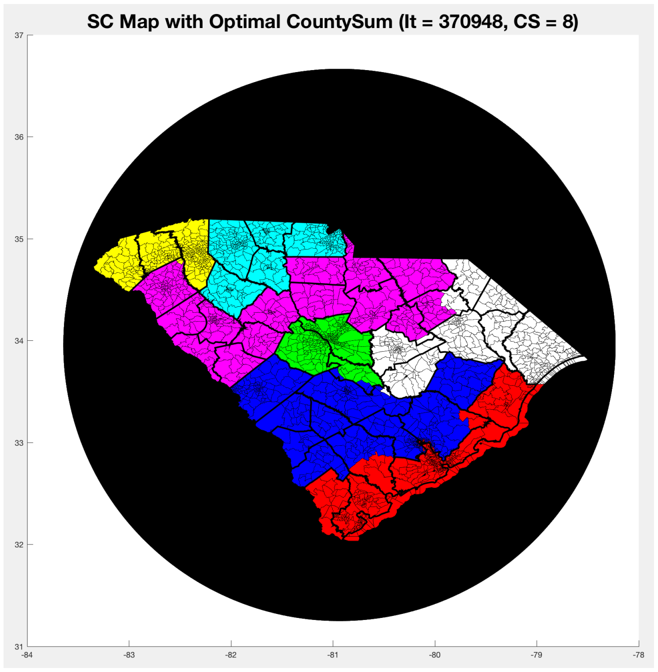


Figure 4.6: The Plan with Fewest County Splits (8), Experiment 4

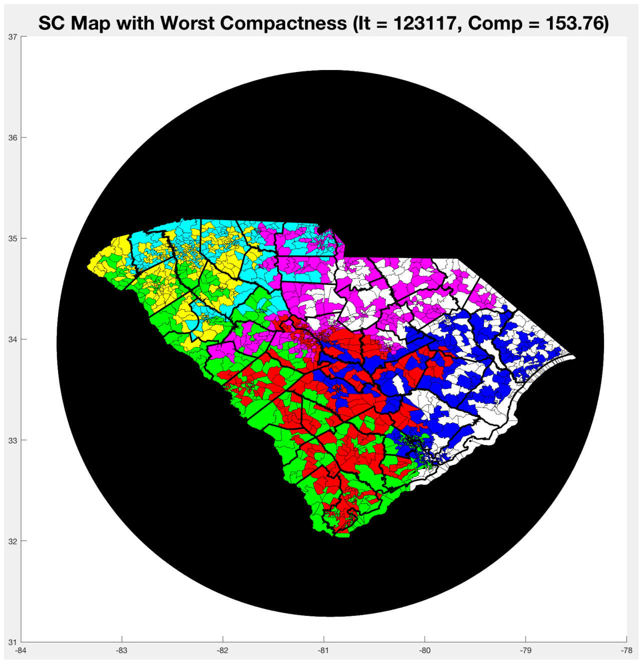


Figure 4.7: Plan with Worst PP_i Compactness Score (152.76), Experiment 2

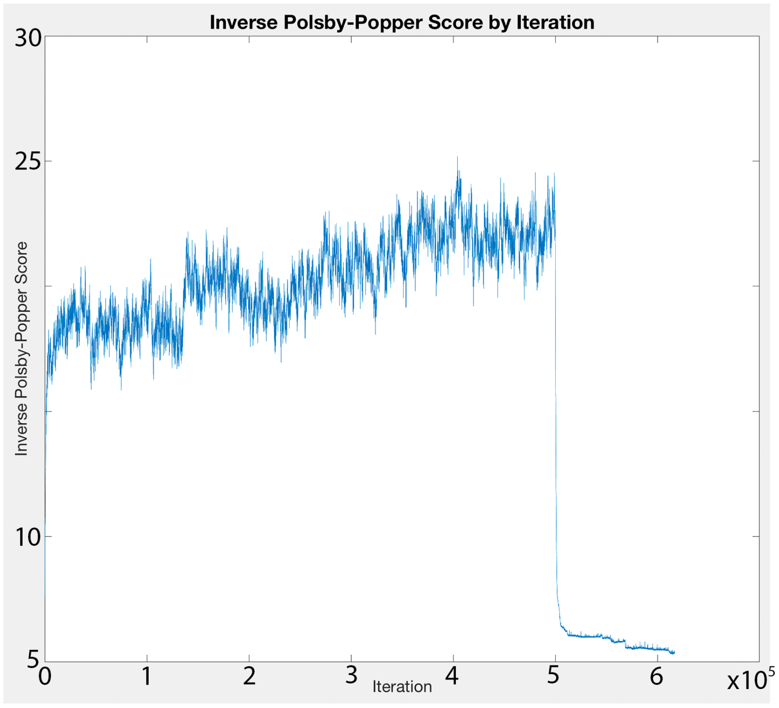


Figure 4.8: Experiment 6 Run 1 Compactness Graph. Cooling Schedule A

The graphs that display compactness scores at each iteration produce vastly different trends based on the cooling schedule. To illustrate the three different cooling schedules, consider the compactness graphs in experiments 6, 3, and 1. These graphs are found respectively in Figures 4.8, 4.9, and 4.10.

4.5 Conclusions

There are a variety of observations we can make about the results produced. To begin, it is notable that in every experiment, we were able to reliably produce plans that are more compact than the 2019 congressional district plan. This provides some evidence that the 2019 plan specifically ignored compactness in an effort to achieve some other goal. Another observation is that if I is set to zero, then predictably, the districts do not conform to county boundaries well, and the number of county splits is large. Also, if L is large, then the districts *never* become excessively noncompact. A mid-algorithm graph with awful compactness is found in Figure 4.7. In the experiments when $L = 50$, the districts tend to migrate their overall shape instead of sprawling all over the state in a spiderweb-like fashion.

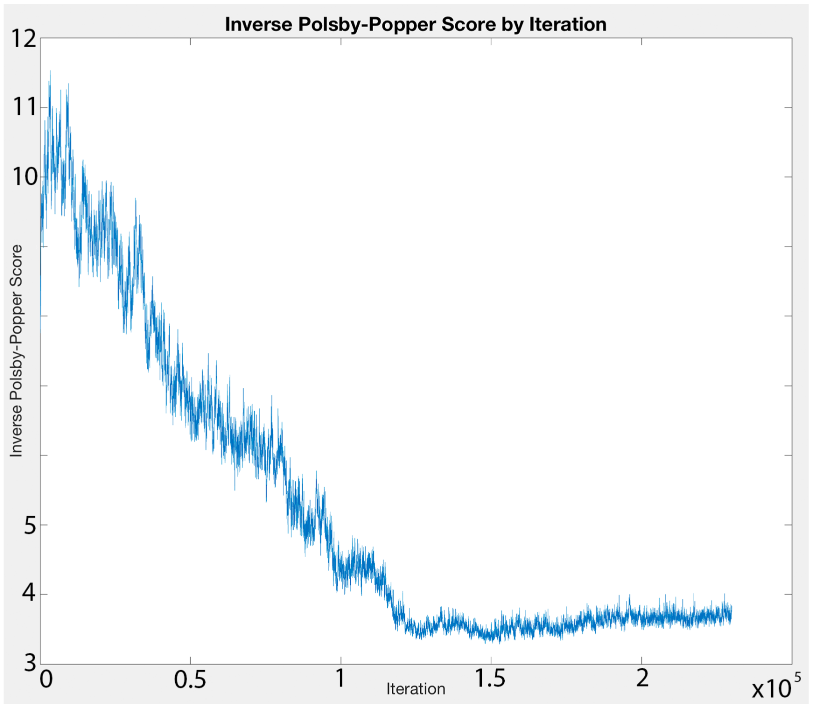


Figure 4.9: Experiment 3 Run 1 Compactness Graph. Cooling Schedule B

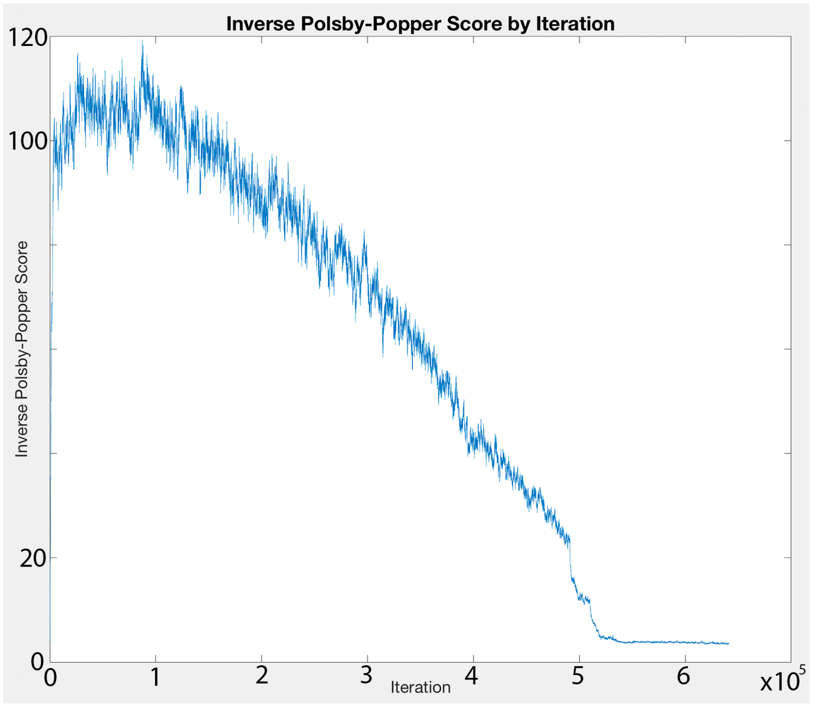


Figure 4.10: Experiment 1 Run 10 Compactness Graph. Cooling Schedule C

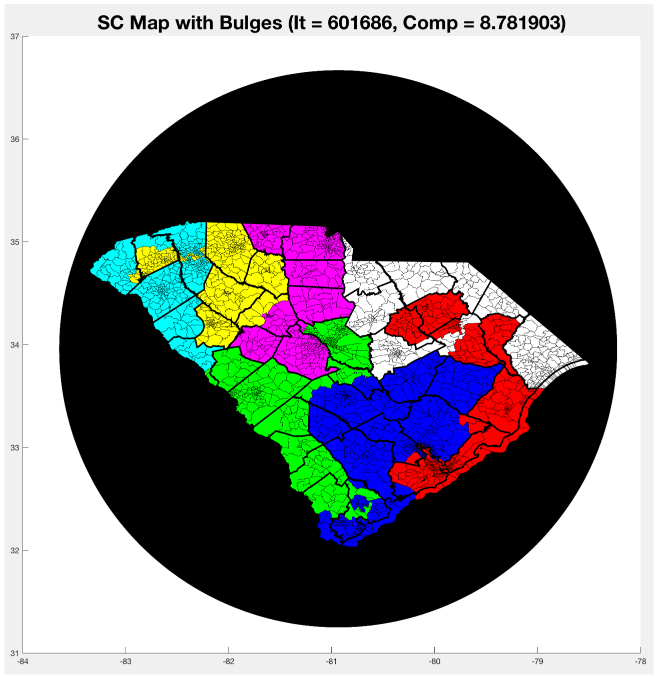


Figure 4.11: Plan with Dumbbell Phenomenon

One surprising result is that deemphasizing county boundaries (setting $I = 0$) does not produce plans that are regularly more compact than experiments that do emphasize county boundaries. One hypothesis to explain this involves the fact that counties are inherently compact shapes. By forcing a district to conform to county lines, we are also forcing the district to be a generally compact shape.

Another notable result is that gradual cooling used in cooling schedules B and C outperformed the instant drop in cooling schedule A. One possible explanation for this is that when the plan is still being heated, it sprawls out. If it is instantly cooled, then the sprawled district may seek to consolidate in two different locations, instead of one, creating a dumbbell-like shape. An example of this phenomenon can be witnessed in Figure 4.11, which came from experiment 9.

Finally, in order to isolate which parameters affected the outcome the most, we average the values in each column of Table 4.4 if the experiment used particular parameter values. This data is reported in Table 4.5. We further analyze how the inputs affected the outputs in Figures 4.12 and 4.13. From these figures, we can conclude that larger values of L lead to better compactness and county split values. We further observe that the binary input I works as intended; when $I = 0$, compactness improves, but county splits worsen. When $I = 1$, county splits improve, but

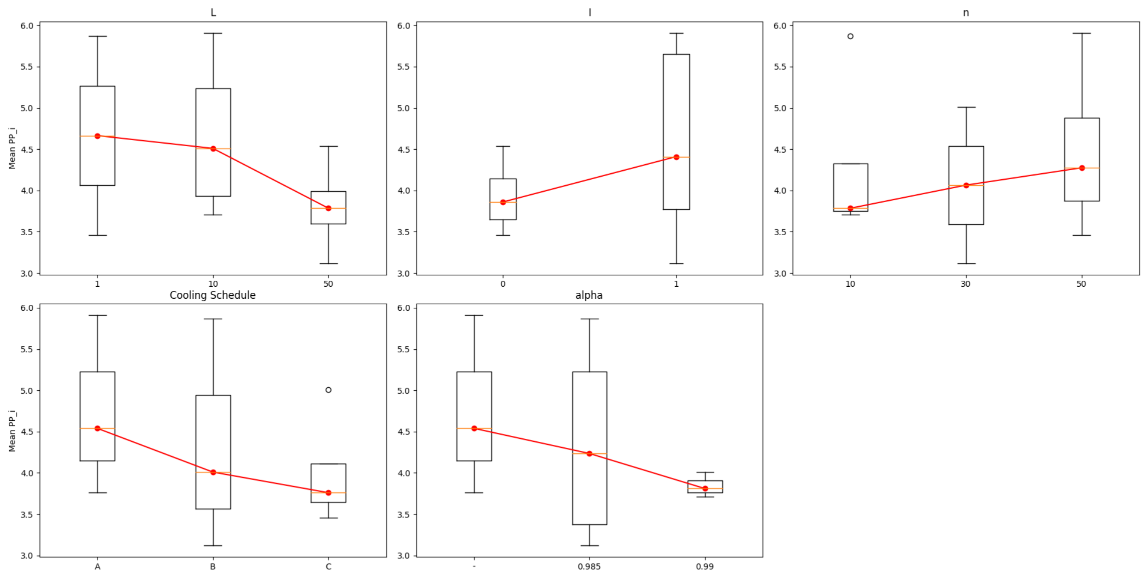


Figure 4.12: Box-and-Whisker Plots for Determining how Inputs affect Compactness

compactness worsens. Further experimentation would likely need to be performed to make definitive conclusions about other inputs (n, cooling schedule, α) affect the final results.

Overall, we demonstrate that simulated annealing is a viable tool for optimizing compactness in South Carolina. Further, we show that modifying the ΔE calculation and the procedure by which perturbations are performed can be an effective technique for improving county splits as well.

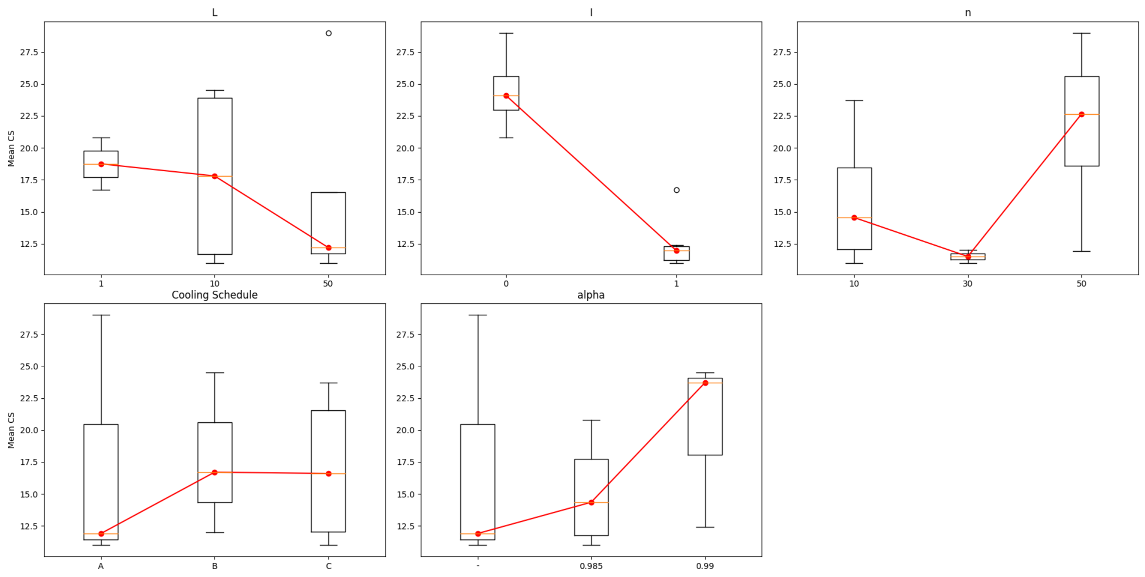


Figure 4.13: Box-and-Whisker Plots for Determining how Inputs affect County Splits

Parameter	Metric to Average					
	Best PP_i	Mean PP_i	Min CS	Mean CS	Worst PP_i	Mean Iterations
$L = 1$	4.20	4.67	16.5	18.8	150.38	539150
$L = 10$	3.58	4.66	14.8	17.8	103.47	511050
$L = 50$	3.49	3.81	13.5	19.1	16.58	449875
$I = 0$	3.44	3.93	21.3	24.5	102.85	566475
$I = 1$	3.82	4.58	10.2	14.5	61.99	442983
$n = 10$	4.00	4.29	13.8	19.0	73.97	494825
$n = 30$	3.11	4.07	9.0	11.5	48.89	397050
$n = 50$	3.62	4.48	18.3	21.6	96.83	587625
CoolSch = A	3.92	4.74	14.3	17.3	46.22	623200
CoolSch = B	3.77	4.33	15.8	17.7	69.77	422567
CoolSch = C	3.40	4.04	14.5	20.0	93.25	446625
α = -	3.92	4.74	14.3	17.3	46.22	623200
$\alpha = 0.985$	3.65	4.37	12.8	15.1	99.63	418075
$\alpha = 0.99$	3.44	3.84	17.3	24.2	81.26	460633

Table 4.5: Average value for each parameter

Chapter 5

MOSA2: Multiobjective Simulated Annealing 2

Although the work in the Chapter 4 does a decent job of optimizing compactness and secondarily minimizing county splits, it lacks the capability to consider other real-world objectives, particularly regarding political fairness. This chapter extends the concept of simulated annealing to deal with even more objectives. Before discussing those objectives, it is worth detouring to discuss multiobjective optimization briefly.

5.1 Multiobjective Optimization

Multiobjective Optimization is the study of simultaneously optimizing two or more objectives subject to one or more constraints. Formally, WLOG, a multiobjective problem can be modeled as:

$$\begin{aligned} \min \quad & [f_1, f_2, \ldots, f_n] \\ \text{S.t.} \quad & [\text{constraints}] \end{aligned} \tag{5.1}$$

where $\{f_i\}_{i=1}^{n}$ are the objective functions. In contrast to single-objective problems, multiobjective problems do not generally have a single solution $\mathbf{x}^*$ that optimizes all objectives simultaneously. Instead, a set of solutions is considered optimal if there does not exist a solution that can improve at least one objective value without worsening any other objective. For instance, consider a two-objective (also known as biobjective) minimization problem, with objectives f_1 and f_2. The optimal set of solutions is known as the *Pareto front* and is deemed to be *Pareto optimal.* Figure 5.1 shows a sample objective function space for such a problem. It is worth noting that the decision variables $\{\boldsymbol{x}^j\}$ that produce the points in Figure 5.1 need not have any particular geometry.

We say that an outcome $\boldsymbol{y}^1$ *dominates* outcome $\boldsymbol{y}^2$ if $\boldsymbol{y}^1_i \leq \boldsymbol{y}^2_i$ for all objectives $i \in [n]$ and that for at least one objective i^*, $\boldsymbol{y}^1_{i^*} < \boldsymbol{y}^2_{i^*}$. We further say that $\boldsymbol{y}^2$ is *dominated* by $\boldsymbol{y}^1$. If neither $\boldsymbol{y}^1$ nor $\boldsymbol{y}^2$ dominates the other, then we say that the two outcomes are *nondominated* with respect to each other. The Pareto front will be the set of outcomes that are nondominated with respect to each other and are not dominated by any other outcome.

Solving multiobjective problems exactly is both possible and well-studied (see for example [8]). In this project, however, due to the sheer quantity of feasible solutions, we focus on metaheuristic techniques. In particular, this chapter utilizes an extension of simulated annealing into multiple dimensions. To assess the efficacy of the approach, we utilize a hypervolume metric.

Hypervolume is a widely used metric for assessing the performance of a set of multiobjective outcomes. The hypervolume indicator measures the volume of the objective space dominated by a given set of outcomes, relative to a reference point. This metric provides a comprehensive assessment by capturing both the convergence and diversity of the outcome set. It is particularly advantageous as it considers the entire Pareto front in every generation, thus offering a scalar value that encapsulates the trade-offs among all objectives. By maximizing the hypervolume, we aim to obtain an outcome set that is not only close to the true Pareto front but also diverse in its coverage of the objective space.

Consider Figure 5.2. The area of the region bounded "northeast" of all outcome points but "southwest" of the reference point represents the hypervolume of this set of outcomes. If a new outcome is found ($\boldsymbol{y}^*$ in the figure) that dominates (or is nondominated wrt) one or more outcomes previously in the set, then the hypervolume increases, since the area grows. As such, any algorithm that gradually finds more nondominated outcomes should have a monotonically increasing hypervolume function.

There are several important notes to make about this calculation:

- This can be easily extended into higher dimensions if there are more than 2 objectives.

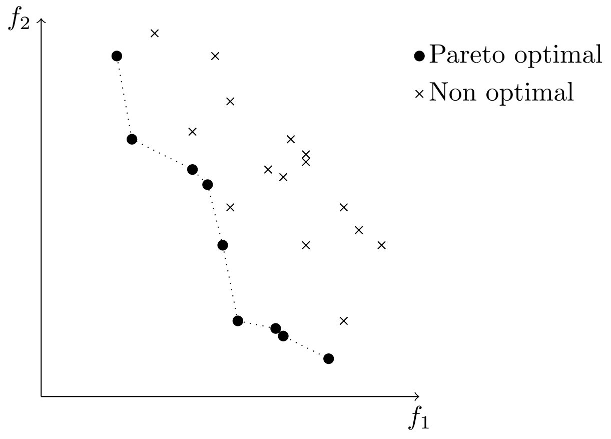


Figure 5.1: Biobjective optimization problem objective function space

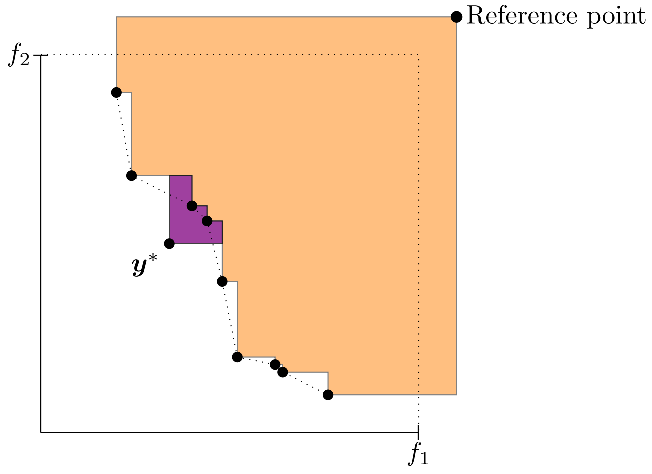


Figure 5.2: Hypervolume for a set of outcomes

- It is important that there is an upper bound for each objective, for without one, no reference point could be reliably plotted. One way to ensure this is to reject any outcomes with objectives greater than some pre-determined upper bound.

- It is wise to scale each objective so that they are all of similar magnitude. For instance, if f_1 had an upper bound of 10^6 and f_2 had an upper bound of 1, then improvements to f_2 would have minimal impact on the hypervolume.

- The choice of a reference point can be impactful. Choosing a reference point that is far northeast of the region would force improvements to hypervolume to be minimal relative to the current area of the region. A standard choice for the reference point is $(1.1, \ldots, 1.1)$ if each objective has upper bound of 1. Ishibuchi discusses additional considerations for choosing a proper reference point [15].

One other concept that is relevant for multiobjective optimization is the idea of an *ideal point.* An ideal point $\boldsymbol{I}$ is the point in the outcome space that presents a theoretical lower bound for each objective. It is improbable that $\boldsymbol{I}$ can be achieved, but it provides a reference for how good the outcomes are.

There are two other metrics that can be helpful in evaluating the quality of a Pareto front. The first metric is the *mean ideal gap* (MIG). The mean ideal gap measures the distance (along each objective) between the ideal point and each point $\boldsymbol{y}$ in the Pareto front, sums these distances together, and divides by the total number of points in the set. Smaller MIG values are better. Equation (5.2) details the precise calculation, where m is the number of points in the Pareto set, n is the number of objectives, $\boldsymbol{y}^j$ is the jth outcome in the set, $\boldsymbol{I}$ is the ideal point, and $\boldsymbol{U}$ is the upper bound vector.

$$MIG = \frac{\sum_{j=1}^{m} \sum_{i=1}^{n} \left(\boldsymbol{y}_i^j - \boldsymbol{I}_i\right) / \boldsymbol{U}_i}{m} \tag{5.2}$$

The second metric, known as *Covered size of space* (CSS) compares two different Pareto sets, by analyzing what percentage of the outcome points dominate outcomes from the other set. Suppose two Pareto sets $\boldsymbol{Y'}$ and $\boldsymbol{Y''}$ are provided. CSS is calculated as defined in equation (5.3), where $\boldsymbol{y'} \in \boldsymbol{Y'}$, and "$\preceq$" means "dominates."[1] A value of $C(\boldsymbol{Y'}, \boldsymbol{Y''}) = 1$ would mean that every outcome in $\boldsymbol{Y''}$ is dominated by an outcome in $\boldsymbol{Y'}$. This calculation will be utilized in Chapter 6.

$$C(\boldsymbol{Y'}, \boldsymbol{Y''}) = \frac{|\{\boldsymbol{y''} \in \boldsymbol{Y''} : \boldsymbol{y'} \preceq \boldsymbol{y''}\}|}{|\boldsymbol{Y''}|} \tag{5.3}$$

5.2 The Algorithm

In this paper, we utilize five objectives that the literature indicates are indicative of a fair districting. Those objectives are (1) population deviation, (2) inverse Polsby-Popper compactness, (3) median-mean fairness score, (4) county splits, and (5) excess geographic units. Additionally, we require that no district is discontiguous. In this study, we first explain one approach for turning

[1]Some authors make the distinction between weak domination ($\preceq$) and strong domination ($\prec$). We omit that distinction here.

simulated annealing into a multiobjective problem, which is by using a weighted objective function. We then explain how our approach (MultiObjective Simulated Annealing - 2) differs.

5.2.1 Perturbations

When creating a plan that gradually changes the shape of the districts, each iteration of the process requires perturbing the districts slightly. There are two primary ways to perturb the districts. The first (and most studied) way of perturbing a districting is to move one GU at a time from one district into a neighboring one. This process, discussed in detail in section 4.2, is known as the 'flip' algorithm. Flips are computationally inexpensive, but can lead to awkwardly shaped districts if done at random.

The second way that districts can be perturbed is through recombination (ReCom), first introduced in [9]. The basic idea behind this technique is to combine two neighboring districts into one, let each GU represent a vertex in a graph, add an edge if GUs share a nonzero length boundary, create a spanning tree on the vertices using Wilson's Algorithm [40], then split the merged districts into two new districts by removing one edge from the spanning tree. ReCom explores the state space Ω more thoroughly by shifting many GUs every iteration, rather than moving them one at a time.

In our algorithm, each individual perturbation utilizes both flips and recombinations. We perform one ReCom step followed by several GU flips. The precise details are explained in subsection 5.2.5.

5.2.2 Weighted-Sum Simulated Annealing

Traditionally, simulated annealing is used in single-objective optimization, as explained in Chapter 4. In order to adjust this algorithm for multiple competing objectives, a weighted sum of the objectives can be computed as follows:

1. Compute all objectives as outlined above. Let the vector of objective values for *plan* be $\boldsymbol{m}(plan)$.

2. Divide the ith objective value by a predetermined scaling factor $\boldsymbol{\lambda}_i$ so that each objective value is similarly scaled.

3. Weight each scaled objective value using a normalized weight vector $\boldsymbol{\alpha}$.

4. The energy for a given plan is calculated with

$$E(plan) = (\boldsymbol{\alpha} \oslash \boldsymbol{\lambda}) \cdot \boldsymbol{m}(plan), \tag{5.4}$$

where "$\oslash$" represents componentwise division and $\cdot$ is the standard dot product. Simulated annealing attempts to locate the plan with the smallest energy, by using the procedure outlined in Algorithm 2. The final output for this algorithm is one plan that is a local minimum for the energy function E. This algorithm is functionally identical to single-objective simulated annealing, except that the energy function is calculated with a weighted sum of multiple objectives. We argue that such an approach leaves much to be desired, though. Our criticisms and alternatives are now presented.

Algorithm 2 Simulated Annealing for Redistricting

```
INPUT:
    $plan_0$: The starting plan
    $T_0$: Initial temperature
    $T_f$: Final temperature
    $its$: Maximum iteration value
    $\boldsymbol{\alpha}$: normalized weight vector
    $\boldsymbol{\lambda}$: scaling vector for objectives
    $\boldsymbol{\lambda}_f$: scaling vector for flips

1: $T \leftarrow T_0$
2: $plan \leftarrow plan_0$
3: $E(plan) \leftarrow (\boldsymbol{\alpha} \oslash \boldsymbol{\lambda}) \cdot \boldsymbol{m}(plan)$        ▷ '$\oslash$' represents componentwise division
4: $\gamma \leftarrow (T_f/T_0)^{1/its}$        ▷ $\gamma$ is the cooling rate for geometric cooling
5: for $i \leftarrow 1$ to $its$ do
6:     $plan' \leftarrow$ perturb($plan$)        ▷ This could be any perturbation function
7:     $E(plan') \leftarrow (\boldsymbol{\alpha} \oslash \boldsymbol{\lambda}) \cdot \boldsymbol{m}(plan')$        ▷ '$\oslash$' represents componentwise division
8:     $\Delta E \leftarrow E(plan') - E(plan)$
9:     if $\Delta E < 0$ then        ▷ i.e., $plan'$ is better than $plan$
10:        $plan \leftarrow plan'$
11:    else        ▷ i.e., $plan'$ is worse than $plan$
12:        $\mu \leftarrow \exp\left(\frac{-\Delta E}{T}\right)$
13:        $plan \leftarrow plan'$ with probability $\mu$
14:    end if
15:    $T \leftarrow \gamma \cdot T$
16: end for
17: return $plan$
```

5.2.3 Multiobjective Algorithm

One criticism of Algorithm 2 is that the final result is a single plan. While this plan may have several desirable characteristics, it is hard to argue that this plan is optimal with regards to every (or even one) objective. In fact, it is likely impossible to find a single plan that optimizes all objectives simultaneously. Even if a diverse set of weights are explored in Algorithm 2, this still may not report the entire approximated Pareto front. An alternative approach is the epsilon-constraint method for multiobjective optimization. However, this approach can be computationally expensive and may require extensive parameter tuning to achieve a satisfactory coverage of the Pareto front, which is impractical for the complexity of redistricting problems. In view of this fact, we implement Algorithm 3, MultiObjective Simulated Annealing 2 (MOSA2). This algorithm is inspired largely from Rincón-García's 2013 paper, which details the invention of the original MOSA for this problem [29]. Our work differs from theirs in a few key ways:

1. We utilize a larger set of objectives. Their paper focused solely on compactness and population deviation.

2. We change the way that our perturbations are performed. Rather than performing flips exclusively, we implement both recombinations and flips, detailed in subsections 5.2.1 and 5.2.5

The end result of MOSA2 is a *set* of high-quality plans that approximate a Pareto front for the objectives considered.

In Algorithm 3, three functions (`generateStartingPlan`, `perturb`, and `saProbCalc`) are used; they are detailed in subsections 5.2.4, 5.2.5, and 5.2.6.

5.2.4 Generate a Starting Plan

Before beginning the algorithm, it is imperative that we choose a plan to serve as the starting point. This is done with the `generateStartingPlan`$(d, \boldsymbol{U})$ function. In this function, d districts are created using a flood-fill technique. In particular, the following steps are performed:

1. Generate a random spanning tree on the GU graph using Wilson's algorithm. Wilson's algorithm is a procedure for generating a random spanning tree on a graph. This procedure has a uniform probability of producing any possible tree for a graph, making it a desirable spanning tree algorithm for this purpose [40].

Algorithm 3 MOSA2: Multiobjective Simulated Annealing 2

```
INPUT:
    $(T_0, T_f)$: (initial temperature, final temperature) pair
    $d$: number of districts to create
    $its$: maximum iteration value
    $\boldsymbol{\lambda}$: scaling vector for objectives
    $\boldsymbol{\lambda}_f$: scaling vector for flips
    $n$: maximum archive size
    $(tol_0, tol_f)$: (initial, final) acceptable tolerance for recombined district populations
    $f$: number of flips to perform per iteration
    $\boldsymbol{U}$: upper bound vector for the objectives
1: initialization:
2:     $plan_0 \leftarrow$ generateStartingPlan($d, \boldsymbol{U}$)          ▷ see subsection 5.2.4
3:     $T \leftarrow T_0$
4:     $tol \leftarrow tol_0$
5:     for $i \leftarrow 1$ to $n$ do
6:         Create a normalized random weight vector (r.w.v.) $\boldsymbol{\alpha}_i$ and add it to a pool $\mathcal{W}$
7:     end for
8:     $plan \leftarrow plan_0$
9:     Archive $\mathcal{A}$ is initialized with maximum size $n$
10:    Add $plan$ to $\mathcal{A}$ and assign it a normalized r.w.v. $\boldsymbol{\alpha}_1$
11:    $\gamma_1 \leftarrow (T_f/T_0)^{1/its}$
12:    $\gamma_2 \leftarrow (tol_f/tol_0)^{1/its}$
13: for $i \leftarrow 1$ to $its$ do
14:    $plan' \leftarrow$ perturb($plan, tol, f, \boldsymbol{\lambda}_f, T$)          ▷ see subsection 5.2.5
15:    if $\exists i : \boldsymbol{m}(plan')_i > \boldsymbol{U}_i$, then
16:        Reject $plan'$ and create a new one. continue.
17:    end if
18:    if $plan'$ dominates at least one plan in $\mathcal{A}$, then
19:        Remove all dominated plans from $\mathcal{A}$
20:        Add $plan'$ to $\mathcal{A}$ and assign it a r.w.v. $\boldsymbol{\alpha}_i$ previously assigned to one of the removed plans
21:        $plan \leftarrow plan'$
22:    else if $plan'$ is dominated by at least one plan in $\mathcal{A}$, then
23:        Do not add $plan'$ to $\mathcal{A}$
24:        $\mu$, $plan_{\mathcal{A}}$, $\boldsymbol{\alpha}_{\mathcal{A}} \leftarrow$ saProbCalc($\mathcal{A}, \boldsymbol{\lambda}, plan', T$)          ▷ see subsection 5.2.6
25:        $plan \leftarrow plan'$ with probability $\mu$, else $plan \leftarrow randomChoice(\mathcal{A})$
26:    else if $plan'$ is nondominated with respect to $\mathcal{A}$ then
27:        if $|\mathcal{A}| < n$ then
28:            Add $plan'$ to $\mathcal{A}$ and assign it a random w.v. $\boldsymbol{\alpha}_i$ from pool $\mathcal{W}$
29:            $plan \leftarrow plan'$
30:        else (i.e. $|\mathcal{A}| = n$)
31:            $\mu$, $plan_{\mathcal{A}}$, $\boldsymbol{\alpha}_{\mathcal{A}} \leftarrow$ saProbCalc($\mathcal{A}, \boldsymbol{\lambda}, plan', T$)          ▷ see subsection 5.2.6
32:            In $\mathcal{A}$, replace $plan_{\mathcal{A}}$ with $plan'$ with prob. $\mu$; if replaced, assign $plan'$ the w.v. $\boldsymbol{\alpha}_{\mathcal{A}}$
33:            $plan \leftarrow plan'$ with probability $\mu$, else $plan \leftarrow randomChoice(\mathcal{A})$
34:        end if
35:    end if
36:    $T \leftarrow \gamma_1 \cdot T$
37:    $tol \leftarrow \gamma_2 \cdot tol$
38: end for
39: return $\mathcal{A}$
```

2. Pick d random GUs to serve as the centers for the districts.

3. For every GU in the tree, add it to the district with the closest center (as measured by graph distance on the tree)

4. If this process produces a plan with objectives that exceed any of the objective bounds in $\boldsymbol{U}$, reject the plan and try again.

Creating a random starting point allows the algorithm to diversify each time the code is run.

5.2.5 Perturb Function

When perturbing a plan, we use one ReCom step and several flips in the following way:

1. In *plan*, pick two neighboring districts d_1 and d_2 with populations p_1 and p_2, respectively. Verify that $p_1 \geq p_I \geq p_2$ where p_I is the ideal population for a district and perform a recombination step for d_1 and d_2. Ensure that the resulting populations (post-ReCom) p_1' and p_2' are within a predetermined tolerance *tol* of the average district population (i.e., $p_1', p_2' \in \left[(1-tol)\left(\frac{p_1+p_2}{2}\right), (1+tol)\left(\frac{p_1+p_2}{2}\right)\right]$). Let the resulting districts be d_1' and d_2'. Over the course of the algorithm, *tol* decreases geometrically to ensure that the district populations are becoming more equinumerous.

2. Perform f flips on the newly created districts, flipping GUs from district d_1' to d_2' or vice-versa, where f is a user-defined input.

   (a) If the flip would create a discontiguous district, reject it. Contiguity is checked using the Python package NetworkX.

   (b) Each flip is judged according to a simulated annealing probability calculation. In particular, we use the following procedure.

      i. Let *plan* and $plan_f$ be the plans before and after the proposed flip, respectively. Let $\boldsymbol{\alpha}$ be the weight vector for both plans, let $\boldsymbol{\lambda}_f$ be the predetermined scaling factor for flips (note that this is distinct from the scaling factor discussed in subsection 5.2.2), and let $\boldsymbol{m}(plan)$ be the function that yields the objectives vector for *plan*.

      ii. Define the energy of the plan as

$$E(plan) = (\boldsymbol{\alpha} \oslash \boldsymbol{\lambda}_f) \cdot \boldsymbol{m}(plan) \tag{5.5}$$

iii. Define the difference of energies as $\Delta E = E(plan_f) - E(plan)$.

iv. If $\Delta E < 0$, then $plan_f$ has a smaller energy, and we accept the change. If $\Delta E \geq 0$, then the flip is accepted with probability

$$\mu = \exp\left(\frac{-\Delta E}{T}\right), \tag{5.6}$$

where T is the current temperature of the algorithm.

It's important to note that $\boldsymbol{\lambda}_f$ should be used instead of $\boldsymbol{\lambda}$ since flips change the objectives very minutely. If $\boldsymbol{\lambda}$ were to be used in equation 5.5, then ΔE in equation 5.6 would be significantly smaller, resulting in the fraction $\frac{-\Delta E}{T}$ being about the same regardless of the actual flip performed. The values for $\boldsymbol{\lambda}_f$ must therefore be significantly smaller than those of $\boldsymbol{\lambda}$ to account for the more minute change caused by flips.

Alternatively, the algorithmic steps are detailed in Algorithm 4.

Algorithm 4 Perturb

```
perturb($plan, tol, f, \boldsymbol{\lambda}_f, T$)
1:  Pick two random neighboring districts $d_1$ and $d_2$ in $plan$ with populations $p_1$ and $p_2$ satisfying
    $p_1 \geq p_I \geq p_2$, where $p_I$ is the ideal population of a district
2:  Perform a ReCom step on $d_1$ and $d_2$, ensuring that the resulting populations $p_1'$ and $p_2'$ both
    satisfy $p_1', p_2' \in \left[(1-tol)\left(\frac{p_1+p_2}{2}\right), (1+tol)\left(\frac{p_1+p_2}{2}\right)\right]$. Define the new districts as $d_1'$ and $d_2'$.
3:  for $j \leftarrow 1$ to $f$ do
4:      Pick a random GU $g_j$ on the border of $d_1'$ and $d_2'$
5:      Let $plan_f$ be the plan after flipping $g_j$ to the neighboring district
6:      if $plan_f$ is discontiguous then
7:          reject the flip
8:      else
9:          $E(plan) \leftarrow (\boldsymbol{\alpha} \oslash \boldsymbol{\lambda}_f) \cdot \boldsymbol{m}(plan)$                ▷ Note that $\boldsymbol{\lambda}_f \neq \boldsymbol{\lambda}$
10:         $E(plan_f) \leftarrow (\boldsymbol{\alpha} \oslash \boldsymbol{\lambda}_f) \cdot \boldsymbol{m}(plan_f)$
11:         $\Delta E \leftarrow E(plan_f) - E(plan)$
12:         if $\Delta E < 0$ then                                   ▷ if the flip improves the plan
13:             $plan \leftarrow plan_f$                                ▷ accept the flip
14:         else                                                  ▷ if the flip worsens the plan
15:             $\mu \leftarrow \exp\left(\frac{-\Delta E}{T}\right)$
16:             $plan \leftarrow plan_f$ with probability $\mu$
17:         end if
18:     end if
19: end for
```

5.2.6 SA Probability Calculation Function

The second function performed in Algorithm 3 is `saProbCalc`$(\mathcal{A}, \boldsymbol{\lambda}, plan', T)$. This function compares a perturbed plan ($plan'$) to a random plan in the archive $\mathcal{A}$ ($plan_{\mathcal{A}}$) and returns three things: (1) a probability μ, (2) the random plan from the archive $plan_{\mathcal{A}}$, and (3) the weight vector $\boldsymbol{\alpha}_{\mathcal{A}}$ assigned to $plan_{\mathcal{A}}$. This value μ is found using the simulated annealing architecture and will be used to determine the likelihood that $plan'$ replaces $plan_{\mathcal{A}}$ in the archive and the likelihood that perturbations will continue to be made on $plan'$. Specifically, the function operates according to Algorithm 5.

Algorithm 5 Simulated Annealing Probability Calculation

```
saProbCalc($\mathcal{A}, \boldsymbol{\lambda}, plan', T$)
1:    $plan_{\mathcal{A}} \leftarrow$ randomChoice($\mathcal{A}$)
2:    $\boldsymbol{\alpha}_{\mathcal{A}} \leftarrow$ weight vector assigned to $plan_{\mathcal{A}}$
3:    $\Delta E \leftarrow (\boldsymbol{\alpha}_{\mathcal{A}} \oslash \boldsymbol{\lambda}) \cdot [\boldsymbol{m}(plan') - \boldsymbol{m}(plan_{\mathcal{A}})]$
4:    $\mu \leftarrow \exp\left(\frac{-\Delta E}{T}\right)$ **if** $\Delta E \geq 0$, **else** $\mu \leftarrow 1$
5:    **Return** $\mu$, $plan_{\mathcal{A}}$, $\boldsymbol{\alpha}_{\mathcal{A}}$
```

5.2.7 Multistart through Parallelism

The algorithm described in Algorithm 3 creates an archive of high-quality plans that approximate a Pareto front for the objectives considered. In order to further diversify this Pareto front and reduce computation time, we run Algorithm 3 several times (excluding the **initialization** steps) using parallel processing so that each start generates a unique output set of plans. We then assess the archive in each parallel process and combine them into one large archive, removing any dominated plans. This process further diversifies the set of solutions and better approximates the Pareto front.

5.3 Experimental Design

This algorithm has a large number of adjustable input parameters, all of which could significantly impact the output. Rather than running hundreds of experiments that would test every possible input combination, we run a small subset of experiments to get a wide breadth of outcomes. These inputs were chosen by using D-optimality (see [22] for details) in an effort to discover which inputs affect the output most significantly. The adjustable input parameters are:

- The various objectives considered in MOSA2
  - Population Deviation (*PD*, always considered)
  - Inverse Polsby-Popper Compactness (PP_i)
  - Median-Mean (*MM*)
  - County Splits (*CS*)
  - Excess geographic units (*EGU*)
- Number of Recombinations performed
- Starting and ending temperature for simulated annealing T_0 and T_f
- Number of flips performed per iteration f
- Archive size n

The fifteen experiments performed are summarized in Table 5.1. In columns PP, MM, CS, and EGU, an X is present if the algorithm uses that objective in its multicriteria calculations. The cell is blank if the objective is ignored. Because of the importance of population equinumerosity in creating districts, we always consider population deviation in our multicriteria calculations. Our code was written in Python 3, and the experiments are performed on Clemson University's supercomputer using ten cores, an Intel Xeon chip, and 120 GB of memory. The codes for this project are found in Appendix C: MOSA2 Codes.

Other input parameters are held constant for each experiment. Those input parameters are located in Table 5.2. The vector $\boldsymbol{\lambda}$ represents the scaling factor for weighted simulated annealing probability calculations. The vector $\boldsymbol{\lambda}_f$ represents the scaling factor for simulated annealing probability calculations, specifically for flips. Finally, $\boldsymbol{U}$ represents the upper bounds for each objective. If any plan is presented that exceeds these bounds, then the plan is discarded.

The values in $\boldsymbol{\lambda}$ and $\boldsymbol{\lambda}_f$ were found experimentally. Roughly speaking, the values in $\boldsymbol{\lambda}$ represent an average change from iteration to iteration for each objective. The values in $\boldsymbol{\lambda}_f$ represent the average change from an arbitrary GU flip. We find that these values generally place each objective in the same order of magnitude. The values of $\boldsymbol{U}$ were also chosen through experimentation.

Thirty trials of each experiment are performed. We then analyze the performance of the algorithm using the hypervolume calculation, discussed in section 5.1. To ensure that the hypervol-

Exp No.	PP	MM	CS	EGU	Recoms	T_0	T_f	Flips	Archive Size
1		X		X	1000	50	0.1	20	250
2	X	X	X	X	2500	20	0.01	50	250
3	X	X	X		500	20	0.01	20	50
4	X	X		X	2500	10	0.005	10	50
5	X			X	1000	20	0.01	50	50
6		X	X		1000	10	0.005	50	50
7			X	X	500	10	0.005	50	125
8	X		X		1000	50	0.1	10	125
9		X	X	X	2500	20	0.01	10	125
10	X		X	X	500	10	0.005	10	250
11	X	X		X	500	50	0.1	50	125
12			X	X	2500	50	0.1	20	50
13	X	X	X	X	500	50	0.1	10	50
14	X	X	X		2500	50	0.1	50	250
15	X				2500	10	0.005	20	125

Table 5.1: Experiments Performed

Input Parameter	PD	PP	MM	CS	EGU
$\boldsymbol{\lambda}$	20000	0.5	0.01	1	25
$\boldsymbol{\lambda}_f$	3000	0.05	0.0001	0.05	1
$\boldsymbol{U}$	2,047,370	9	0.05	50	500

Table 5.2: Input Parameters for MOSA2

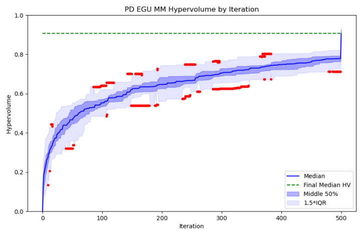
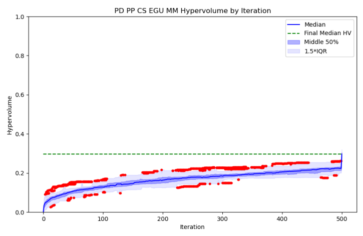
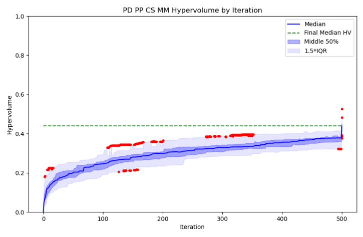
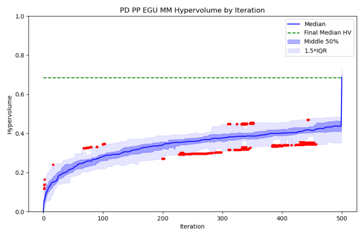
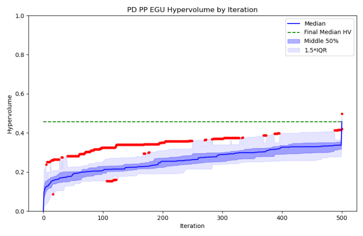
ume measures each objective similarly, all objectives are scaled by their upper bounds in $\boldsymbol{U}$, and the reference point is placed at $(1.1, \ldots, 1.1)$.

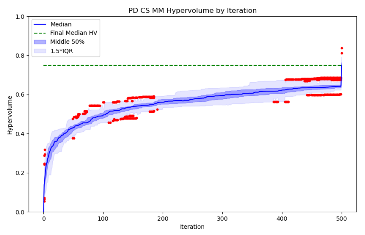
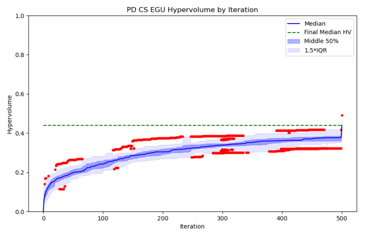
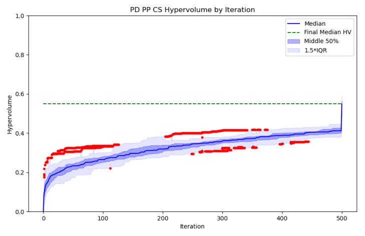
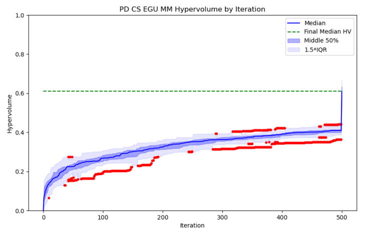
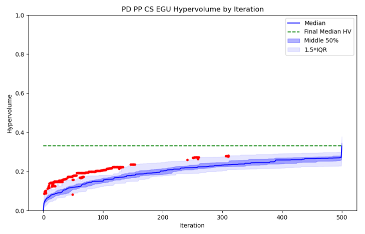
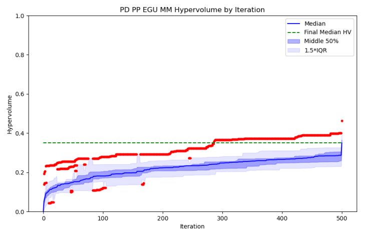
5.4 MOSA2 Results

The results of each experiment are detailed in Table 5.3. Since each experiment ran 30 trials, and each trial utilized parallel processing, creating the hypervolume graphs required some ingenuity. Specifically, the hypervolume graphs depicted in the table were created by performing the following steps for each experiment:

- Calculate the median hypervolume value amongst the five threads of the parallel process for every iteration. For iteration i, call this value h^i_{med}.
- For each iteration i, collect the set of h^i_{med} values for each of the 30 trials performed. For iteration i, call this set $\mathcal{H}_i$.

- Find the median, 25th percentile, and 75th percentile values for $\mathcal{H}_i$.
- Plot the values of $\mathcal{H}_i$ by doing the following:
  - let a dark line represent the median of $\mathcal{H}_i$ for each iteration i,
  - let a dark band around the median represent the extent between the 25th and 75th percentiles,
  - let a lighter band around the dark band extend to the largest and smallest values in $\mathcal{H}_i \cap \left[\ell - \frac{3}{2}IQR, u + \frac{3}{2}IQR\right]$ where ℓ is the 25th percentile value, u is the 75th percentile value, and IQR is the interquartile range,
  - plot red dots for any outliers in $\mathcal{H}_i \setminus \left[\ell - \frac{3}{2}IQR, u + \frac{3}{2}IQR\right]$.
- At the end of the algorithm, all parallel processes combine their archives of plans, and inevitably, the hypervolume increases. Plot this final median hypervolume as a dotted horizontal line to illustrate the median hypervolume achieved from the 30 trials.

Exp	Inputs				Hypervolume Graph	Final Median HV
1	PD:	X	ReComs:	1000		0.906
	PP:		T_0:	50		
	MM:	X	T_f:	0.1		
	CS:		Flips:	20		
	EGU:	X	n:	250		
2	PD:	X	ReComs:	2500		0.293
	PP:	X	T_0:	20		
	MM:	X	T_f:	0.01		
	CS:	X	Flips:	50		
	EGU:	X	n:	250		
3	PD:	X	ReComs:	500		0.415
	PP:	X	T_0:	20		
	MM:	X	T_f:	0.01		
	CS:	X	Flips:	20		
	EGU:		n:	50		
4	PD:	X	ReComs:	2500		0.685
	PP:	X	T_0:	10		
	MM:		T_f:	0.005		
	CS:		Flips:	10		
	EGU:	X	n:	50		
5	PD:	X	ReComs:	1000		0.457
	PP:	X	T_0:	20		
	MM:		T_f:	0.01		
	CS:		Flips:	50		
	EGU:	X	n:	50		

6	PD:	X	ReComs:	1000		0.750
	PP:		T_0:	10		
	MM:	X	T_f:	0.005		
	CS:	X	Flips:	50		
	EGU:		n:	50		
7	PD:	X	ReComs:	500		0.438
	PP:		T_0:	10		
	MM:		T_f:	0.005		
	CS:	X	Flips:	50		
	EGU:	X	n:	125		
8	PD:	X	ReComs:	1000		0.549
	PP:	X	T_0:	50		
	MM:		T_f:	0.1		
	CS:	X	Flips:	10		
	EGU:		n:	125		
9	PD:	X	ReComs:	2500		0.610
	PP:		T_0:	20		
	MM:	X	T_f:	0.01		
	CS:	X	Flips:	10		
	EGU:	X	n:	125		
10	PD:	X	ReComs:	500		0.331
	PP:	X	T_0:	10		
	MM:		T_f:	0.005		
	CS:	X	Flips:	10		
	EGU:	X	n:	250		
11	PD:	X	ReComs:	500		0.350
	PP:	X	T_0:	50		
	MM:	X	T_f:	0.1		
	CS:		Flips:	50		
	EGU:	X	n:	125		

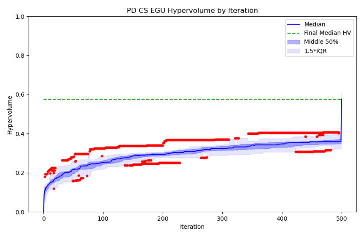
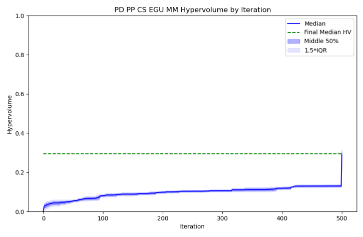
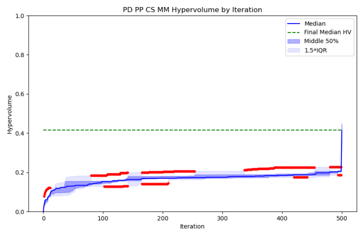
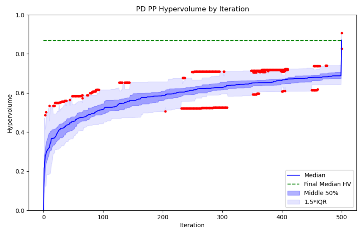
	PD:	X	ReComs:	2500		
	PP:		T_0:	50		
12	MM:		T_f:	0.1		0.576
	CS:	X	Flips:	20		
	EGU:	X	n:	50		
	PD:	X	ReComs:	500		
	PP:	X	T_0:	50		
13	MM:	X	T_f:	0.1		0.297
	CS:	X	Flips:	10		
	EGU:	X	n:	50		
	PD:	X	ReComs:	2500		
	PP:	X	T_0:	50		
14	MM:	X	T_f:	0.1		0.439
	CS:	X	Flips:	50		
	EGU:		n:	250		
	PD:	X	ReComs:	2500		
	PP:	X	T_0:	10		
15	MM:		T_f:	0.005		0.868
	CS:		Flips:	20		
	EGU:		n:	125		

Table 5.3: Summary of Hypervolume Graphs for Each Experiment

We also present the mean ideal gap metric for these experiments, with an ideal point described in Table 5.4. The source for the various ideal point values comes either from work previously mentioned in this dissertation or from outside literature. Figure 5.3 displays the box-and-whisker plots for each experiment, where the data points come from the various trials. As mentioned in section 5.1, smaller values for MIG are preferable, since this indicates that the outcome plans have objective values close to the ideal point. As we can see, the MIG values for experiments 1 and 15 are smallest, which follows closely from the hypervolume graphs, since those experiments produced

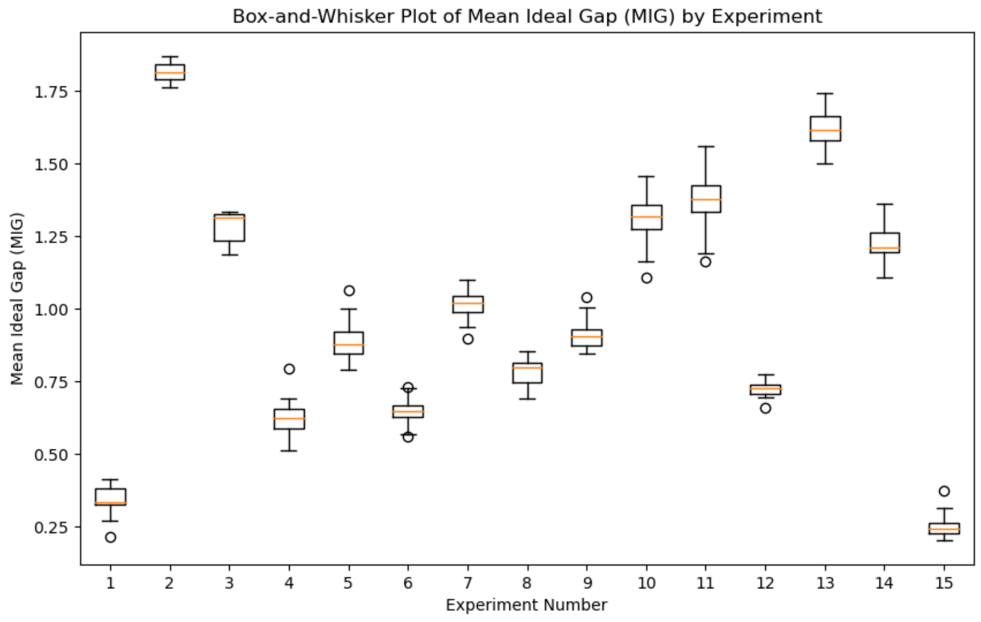


Figure 5.3: Mean Ideal Gap Values for Experiments in MOSA2

the largest hypervolumes. The plan with the best compactness score can be found in Figure 5.4 and emerged from experiment 15. This plan has a population deviation of 37949, representing less than a 1% difference from ideal.

Obj	$\boldsymbol{I}$	Source
PD	9	Chapter 3
PP_i	1.65	Chapter 4
MM	0	Chapter 3
CS	1	Shahmizad and Buchanan [30]
EGU	91	Shahmizad and Buchanan [30]

Table 5.4: Ideal Point Values and their Sources

We also report the box-and-whisker plots that detail how inputs affect the hypervolume for these parameters as well. Those plots are reported in Figure 5.5. In each of these plots, the y-axis is hypervolume, and the x-axis reports the various inputs utilized for the experiments. The most striking result seen here is that hypervolume improves when fewer objectives are considered. In the first four plots, hypervolume is generally lower when each objective is utilized, and is higher when the individual objective is not considered.

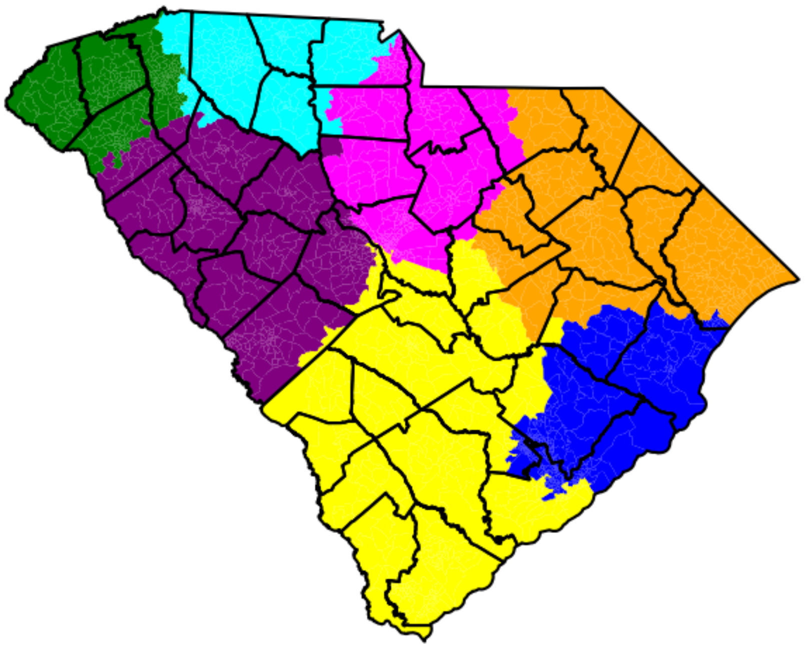


Figure 5.4: MOSA2 Plan with Best Compactness ($PD = 37949, PP_i = 2.745$). Experiment 15.

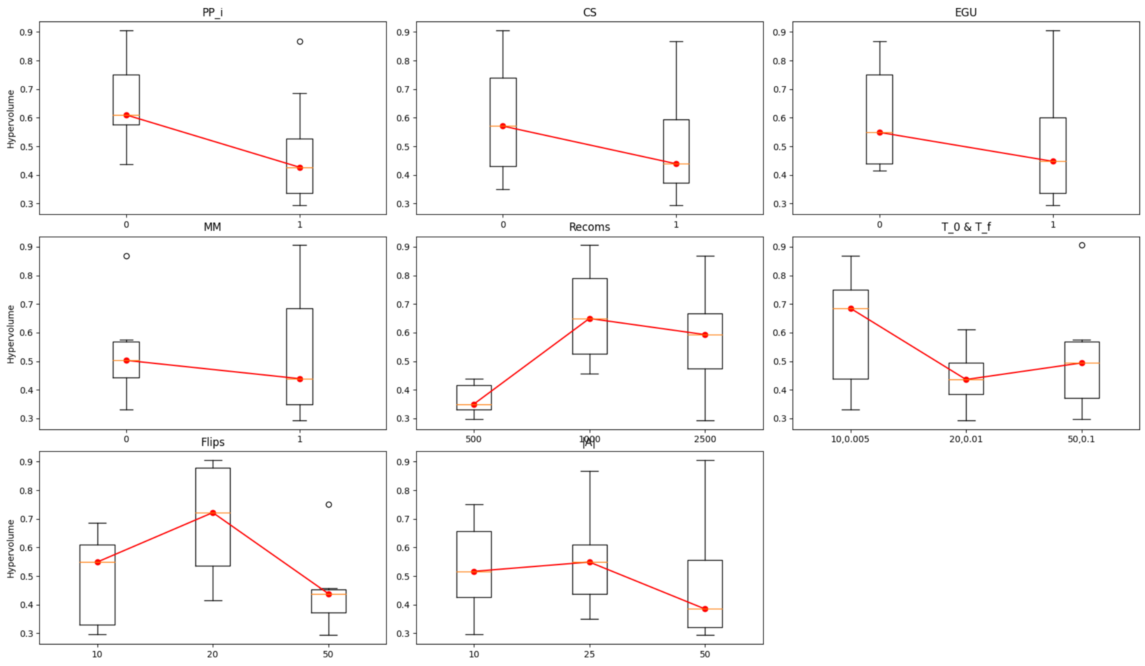


Figure 5.5: MOSA2 Box-and-Whisker Plots for Assessing Input Parameter Effects

Overall, we find that MOSA2 produces quality plans, especially when the number of objectives is small, and when the number of iterations (ReComs) is large. This work represents a more robust generalization of simulated annealing than the formulation presented in Chapter 4, since more objectives can be easily considered in this work. We do, however, find better compactness results in Chapter 4, suggesting that prior work still has merit if political fairness measures are to be ignored. Future work for this project might include the following ideas:

- Changing how objective scales ($\boldsymbol{\lambda}$ and $\boldsymbol{\lambda}_f$) are chosen. This experiment chose these values in a very ad hoc way, so a more systematic approach may be fruitful. It also remains to be seen what impact choosing these scales has on the final plans produced.
- Adding in other objectives (such as EG, which was omitted here) may provide other interesting results.
- It would be interesting to explore how much impact the addition of ReComs in the algorithm improved performance compared to the original algorithm developed by Rincón-García [29].

Chapter 6

NSGA-II for Political Redistricting

6.1 Introduction

Having adapted a traditionally single-objective problem (SA) into a multiobjective problem (MOSA), we now divert our attention to procedures designed explicitly for multiple objectives. In particular, we study the multiobjective algorithm of the Non-Dominated Sorting Genetic Algorithm-II, or NSGA-II for short. We adapt a previous approach taken by Vanneschi et al. to get better hypervolume results. We also approach this problem by considering more objectives than prior studies have undertaken.

The objectives we have selected for this project are:

1. Population Deviation (PD)
2. Shifted Polsby-Popper Compactness (PP_s)
3. County Splits (CS)
4. Efficiency Gap (EG)
5. Median Mean (MM)

NSGA-II breeds good solutions with one another to produce other high-quality solutions [7]. Doing this procedure for many generations has shown to be a good metaheuristic strategy for solving many complex problems. Our work in this paper follows many of the ideas introduced by Vanneschi et al. in [36]. The novelty in our work emerges on two fronts: (1) the number of objectives considered is larger than that of previous works and (2) the crossover step of NSGA-II utilizes a novel clustering approach which is better tailored for the context of genetic algorithms.

This study is divided into the following sections. Section 6.2 describes the algorithm used in this paper and the experimental design employed. Section 6.3 then displays the results of the experiments and section 6.4 concludes the paper and discusses potential future work.

6.2 Methods

6.2.1 Previous Approaches

Vanneschi et al.'s algorithm uses NSGA-II and Variable Neighborhood Search to perform political redistricting. For brevity, we refer to Vanneschi's algorithm as VNSGA-II henceforth. VNSGA-II is summarized in Algorithm 6.

In general, the philosophy behind genetic algorithms is that if you select two solutions to be parents, combining the "genes" of the parents oftentimes produces even better children. In the algorithm above (and all other NSGA-II redistricting works to our knowledge) we notice that there is nothing inherently valuable about the genes of the parents. In the example from line 2 in Algorthm 6, there is no reason to believe that the 225th GU being assigned to district 2 influences (positively or negatively) the quality of the solution. However, we argue that there *is* inherent value in determining which GUs should be clustered together in the same districts.

To illustrate our point, consider the example graphs in Figure 6.1. Suppose each node represents a GU, and an edge is present if the GUs have a nonzero-length boundary. The color of the nodes represents the districts to which they are assigned. Letting the blue (star pattern) nodes correspond to district 1, red (diagonal line pattern) nodes correspond to district 2, and yellow (patternless) nodes correspond to district 3, we can encode the parents as arrays:

$$s_1 = \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|}\hline 1 & 1 & 2 & 1 & 3 & 2 & 3 & 3 & 2 & 2 & 3 & 3 & 2 & 2 & 2 \\ \hline \multicolumn{1}{c}{1} & \multicolumn{1}{c}{2} & \multicolumn{1}{c}{3} & \multicolumn{1}{c}{4} & \multicolumn{1}{c}{5} & \multicolumn{1}{c}{6} & \multicolumn{1}{c}{7} & \multicolumn{1}{c}{8} & \multicolumn{1}{c}{9} & \multicolumn{1}{c}{10} & \multicolumn{1}{c}{11} & \multicolumn{1}{c}{12} & \multicolumn{1}{c}{13} & \multicolumn{1}{c}{14} & \multicolumn{1}{c}{15} \end{array}$$

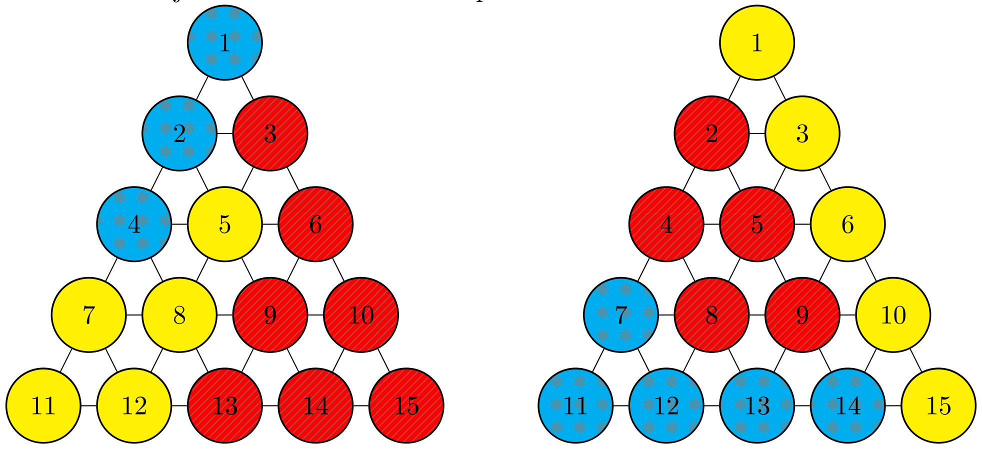
$$s_2 = \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|}\hline 3 & 2 & 3 & 2 & 2 & 3 & 1 & 2 & 2 & 3 & 1 & 1 & 1 & 1 & 3 \\ \hline \multicolumn{1}{c}{1} & \multicolumn{1}{c}{2} & \multicolumn{1}{c}{3} & \multicolumn{1}{c}{4} & \multicolumn{1}{c}{5} & \multicolumn{1}{c}{6} & \multicolumn{1}{c}{7} & \multicolumn{1}{c}{8} & \multicolumn{1}{c}{9} & \multicolumn{1}{c}{10} & \multicolumn{1}{c}{11} & \multicolumn{1}{c}{12} & \multicolumn{1}{c}{13} & \multicolumn{1}{c}{14} & \multicolumn{1}{c}{15} \end{array}$$

If we arbitrarily set the crossover point to be 3, then to create child 1 s_1', we iteratively check through the indices $i = 1, 2, 3$ from s_1 and see if they can change district to $s_2[i]$ without violating

Algorithm 6 VNSGA-II for Political Redistricting [36]

```
Input:
    $n$: population size
    $g$: number of generations to run algorithm
    $G = (V, E)$: GU graph with vertex set $V$ and edge set $E$
    $d$: number of districts to create
    $t$: number of solutions to choose for tournament selection
    $\mu$: mutation probability
    $p_{VNS}$: probability to use variable neighborhood search
 1: Randomly generate $n$ feasible plans and add them to a set $P$.
 2: Encode each plan as an array $s$ with length equal to the number of GUs. The $i$th element of
    this array describes the district to which the $i$th GU is assigned. For example, $s[225] = 2$ would
    mean that the 225th GU is assigned to district 2.
 3: while $gen < g$ do
 4:     Create empty set $P'$ which will contain the children of this generation.
 5:     while $|P'| < n$ do
 6:         if $|P'| < 0.6n$ then perform crossover:
 7:             Select two plans $s_1$ and $s_2$ in $P$ to be the parents using tournament selection (described
                in lines 8 - 12).
 8:             for each parent do
 9:                 Randomly select $t$ solutions from $P$.
10:                 Amongst those $t$ solutions, pick the solution(s) with the lowest rank (closest to the
                    best front)
11:                 If more than one solution is selected, choose the one closest to the median solution
                    of the front.
12:             end for
13:             Breed these two parents together to create two children $s'_1$ and $s'_2$.
14:                  For some GU $i$, let $s_1[i] = \mathcal{D}_1$ and $s_2[i] = \mathcal{D}_2$.
15:                  Pick a random crossover point $c$ (i.e., pick an integer between 1 and $|V|$)
16:                  Breed child 1: $s'_1 := s_1$ except for each GU $i \leq c$, $s'_1[i] := \mathcal{D}_2$ if contiguity in $s'_1$ is
                     not violated.
17:                  Breed child 2: $s'_2 := s2$ except for each GU $i > c$, $s'_2[i] := \mathcal{D}_1$ if contiguity in $s'_2$ is
                     not violated.
18:                  Add $s'_1$ and $s'_2$ to $P'$.
19:         else if $|P'| < 0.9n$ then perform mutation:
20:             Randomly select a plan $s \in P$. Set $s' := s$.
21:             In $s'$, randomly swap up to $\mu|V|$ GUs to neighboring districts, ensuring contiguity
                isn't violated.
22:             Add $s'$ to $|P'|$.
23:         else perform copying:
24:             Randomly select a plan $s \in P$. Add $s$ to $|P'|$.
25:         end if
26:     end while
27:     Perform a variable neighborhood search (VNS) on $p_{VNS} \cdot n$ plans.
28:     Amongst all plans in $P \cup P'$, build Pareto fronts according to the objective evaluations.
29:     Calculate the crowding distance for each of the plans.
30:     Add the best $n$ plans to a set $P''$. Empty $P$ and $P'$. $P \leftarrow P''$. Empty $P''$.
31:     $gen \leftarrow gen + 1$
32: end while
```

Figure 6.1: An example of two potential parents. Each node represents a GU and each edge signifies that those GUs are adjacent. Node colors correspond to different districts.



contiguity. More precisely, we observe that:

- $s_1[1] = 1$ currently. We check to see if changing to $s_1[1] = 3$ causes any discontiguity. From Figure 6.1, we notice that node 1 cannot turn yellow, else the yellow district would be discontiguous.

- $s_1[2] = 1$ currently. We check to see if changing to $s_1[2] = 2$ causes any discontiguity. From Figure 6.1, we notice that node 2 cannot turn red, as the blue district would be discontiguous.

- $s_1[3] = 2$ currently. We check to see if changing to $s_1[3] = 3$ causes any discontiguity. From Figure 6.1, it does not create a discontiguity, so s_1' is identical to s_1 except $s_1[3] = 3$.

The issue here is that s_1' is not 'inheriting' any traits from s_2. Most NSGA-II efforts are effectively implementing arbitrary GU flips to generate new plans. Even if the objectives evaluated for both parents are strong, there is no reason to believe that this crossover generates children with strong objectives themselves.

VNSGA-II utilizes a variable neighborhood search (VNS) in line 27 of Algorithm 6. This is where most of the plan improvement occurs. To summarize, VNS seeks to optimize a plan by repeatedly applying hill climbing with respect to different neighborhood structures $\{N_i\}_{i=1}^4$. Each structure defines the type of perturbation that can be performed to get from one plan to another. For example, structure N_1 allows perturbations that flip a GU from one district to a neighboring district if population balance is improved and district contiguity is maintained. With predetermined probability p_{VNS}, VNS performs the following steps on a plan s for each neighbor structure N_i:

1. Select a solution s from the current population P.

2. Generate a solution s' by making a legal perturbation according to N_i.

3. Apply hill climbing on s' using legal perturbations according to N_i to create s''.

4. If s'' dominates s with respect to *all objectives*, set $s := s''$.

6.2.2 Our Approach

In contrast to prior works, we use a novel clustering approach which better models the theme of genetic algorithms. Broadly speaking, we group several neighboring GUs together into 'clusters' and use those clusters as building blocks for the districts. The fundamental premise of this approach is that if a cluster of GUs helps to create a 'good' district, it survives from generation to generation. A rough sketch of our approach can be found in Algorithm 7.

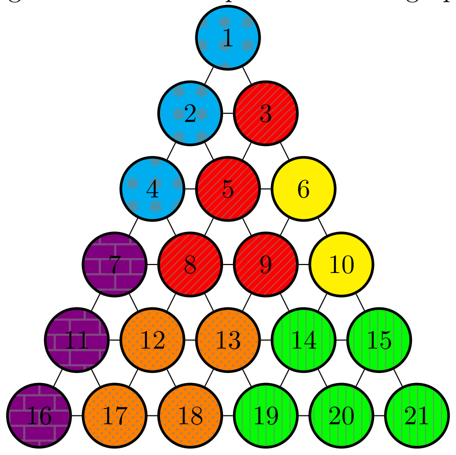
Algorithm 7 Our Approach to NSGA-II

```
Input:
    $g$: Number of generations
    $\mu$: Mutation probability
    $n$: Population size
    $d$: Number of districts
    The upper bounds for each objective
    $G = (V, E)$: GUI graph with vertex set $V$ and edge set $E$
1: Build $n$ random clusters on the graph $G$.                          ▷ See subsection 6.2.2.1
2: Build $n$ districtings on these clusters
3: $gen \leftarrow 0$
4: while $gen < g$ do
5:     Crossover. Breed $n$ child cluster graphs using a crossover path.   ▷ See subsection 6.2.2.2
6:     Mutation. Mutate $\mu n$ children.                                ▷ See subsection 6.2.2.3
7:     Create district plans on these child cluster graphs.              ▷ See subsection 6.2.2.4
8:     Evaluate the objectives on all $2n$ plans in generation $gen$.
9:     Sort the $2n$ plans into ranked fronts, with the rank 1 front being the current approximated
       Pareto Front.
10:    Assign the best $n$ plans (according to front rank) to be the potential parents of the next
       generation. Use crowding distance to determine the worst entrants. ▷ See subsection 6.2.2.5
11:    $gen \leftarrow gen + 1$
12: end while
```

6.2.2.1 Clustering

Our approach utilizes a novel clustering approach, where we cluster several nearby GUs together to form the building blocks for districts. If this grouping of GUs helps to create quality

Figure 6.2: An example of a cluster graph



districts, these clusters are preserved generation to generation. The motivation for this approach emerges from the observation that it is much easier to create compact districts if the underlying building blocks (i.e., the clusters) are compact as well.

To demonstrate a clustering example, consider the graph in Figure 6.2. There are six clusters in Figure 6.2, each represented by a unique color. In our code, clusters are constructed with anywhere between one and six GUs. The districts are then built using these clusters. For example, if the cluster graph in Figure 6.2 has three districts, then those districts may contain the following GUs:

- District 1: GUs $1, 2, 3, 4, 5, 8, 9$
- District 2: GUs $6, 10, 14, 15, 19, 20, 21$
- District 3: GUs $7, 11, 12, 13, 16, 17, 18$

In this algorithm, clustering graphs are created using the procedure outlined in Algorithm 8. The process by which districts are created on the underlying clusters is explained further in subsection 6.2.2.4.

6.2.2.2 Crossover

When performing the crossover step of NSGA-II, we start by randomly selecting two parents s_1 and s_2 from our current generation. Each parent is a clustering—that is, an assignment of GUs

Algorithm 8 Building a Cluster Graph

```
Input:
    GU graph $G = (V, E)$ with vertex set $V$ and edge set $E$
1: while there exists an unassigned GU do
2:     Choose a random unassigned GU to serve as the center for a new cluster $\mathcal{C}$.
3:     Assign this cluster $\mathcal{C}$ a random maximum size $M_{\mathcal{C}}$ between 1 and 6.
4:     while the number of nodes in $\mathcal{C}$ is less than $M_{\mathcal{C}}$ do
5:         Find all unassigned neighboring GUs of $\mathcal{C}$ and choose one randomly* to add to this cluster.
6:         *If county splits is an objective considered in the algorithm, then adding a GU from a
           different county to the current cluster happens with low probability (set here as 1%).
7:         If there are no unassigned neighboring GUs of $\mathcal{C}$, adjust $M_{\mathcal{C}}$ to be the current size of the
           cluster.
8:     end while
9: end while
```

to clusters (as explained in subsection 6.2.2.1). Then, on the underlying graph G, we split G into two subgraphs H_1 and H_2 in the following way:

1. Choose a random boundary GU u (a GU that has nonzero boundary length with the border of the state).

2. Determine the shortest graph distance ℓ_{uv} between GU u and every other boundary GU v.

3. Assign a weight w_{uv} to each (u, v) pair, where $w_{uv} = \ell_{uv}^2$ (the square of the shortest graph distance between u and v).

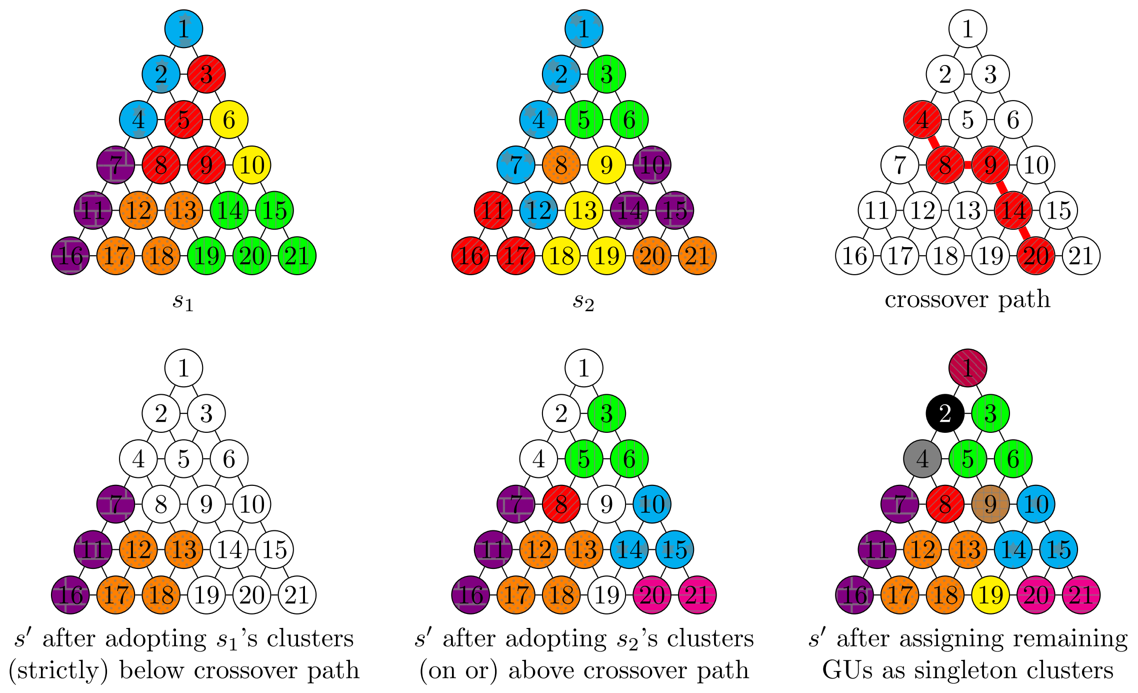
4. Using a random weighted selection, pick v. Let the shortest path between u and v be described as the *crossover path*.

5. Let H_1 and H_2 be the subgraphs of G on either side of the crossover path. Randomly choose either H_1 or H_2 to include all nodes in the crossover path. If the number of subgraphs is not exactly two when extracting the nodes of the crossover path from G, reselect u and v from step 1.

The weighting performed in step 3 is done to incentivize larger subgraphs to be created. If one subgraph is much smaller than the other, then the genes of the parents mix less, which we seek to avoid.

Now, for each cluster $\mathcal{C}$ in s_1, if $\mathcal{C}$ is a subgraph of H_1 (i.e., $\mathcal{C} \subseteq H_1$), then the child s' adopts cluster $\mathcal{C}$. Similarly, for each cluster $\mathcal{C}$ in s_2, if $\mathcal{C} \subseteq H_2$, then s' adopts cluster $\mathcal{C}$. For any GUs that are left unclustered in s', assign them to be singleton clusters. To demonstrate this concept,

Figure 6.3: Crossover steps


s_1
s_2
crossover path
s' after adopting s_1's clusters (strictly) below crossover path
s' after adopting s_2's clusters (on or) above crossover path
s' after assigning remaining GUs as singleton clusters

consider Figure 6.3. The parents s_1 and s_2 breed a child s' by splitting their clusters along the crossover path. H_2 was randomly chosen to contain all nodes in and above the crossover path, while H_1 contains all nodes below the crossover path. In any given generation, n children are created from $2n$ randomly chosen parents in the generation, where n is the population size. Therefore, in any given generation, there are $2n$ cluster graphs (n parents and n children).

6.2.2.3 Mutation

In our algorithm, mutation occurs in several ways. Denote the mutation probability as μ and the population size as n. After the n children are bred using crossover, we mutate μn children (rounding to the nearest integer). There are three operations that are performed on the children:

1. Merging a GU into a neighboring cluster that it is not currently a part of.

2. Splitting a GU that is currently part of a cluster into a singleton cluster.

3. Using recombination to recreate two districts.

Regarding the first two operations, if a child has been designated as one to be mutated, we then randomly pick $\lfloor \mu |V| \rfloor$ of the GUs to mutate (where V is the set of GUs in the graph G). For each GU designated for mutation, we randomly decide whether it is merged into a neighboring cluster or whether it is split into a singleton cluster. If the GU selected is already a singleton cluster, then it is merged into a neighboring cluster. Each time a GU is selected, we first must check whether removing the GU from the cluster causes the cluster to become discontiguous. If so, then we skip that mutation. Further, if county splits are considered as an objective, then merges that create clusters crossing county lines are prohibited.

The third operation occurs only after districts are created. See subsection 6.2.2.4 for details.

6.2.2.4 District Plans

After the mutation step, we then create district plans for the n children. The process by which districts are constructed emphasizes population equality and compactness. For each child, d districts are created using the procedure outlined in Algorithm 9. Each district would ideally have a population of $\mathcal{I} := \frac{\mathcal{P}}{d}$, where $\mathcal{P}$ is the total population of the state. Algorithm 9 tries to encourage such districts to be generated, while also building districts in a compact manner.

This algorithm also rejects district plans where the objectives are too large. Those upper bounds can be specified by the user.

Population equality is encouraged because as the districts grow, the districts with lower population have a higher probability of being chosen to expand. Compactness is emphasized by weighting the surrounding clusters proportional to their boundary length with the district. This procedure makes it less likely that small offshoots from the district emerge.

This code requires that each district have a population between $0.15\mathcal{I}$ and $2\mathcal{I}\alpha^{gen}$, where $\mathcal{I}$ is the ideal population of a district, gen is the current generation, and α is a scaling factor slightly less than 1 such that $2\mathcal{I}\alpha^{g} = 1.5\mathcal{I}$ (where g is the number of generations that are used). These bounds encourage the population of the districts to slowly approach $\mathcal{I}$ over the course of the algorithm. These values, along with the number of sets of centers used (5) were all found experimentally; they balance computation time and algorithmic output.

After districts are created, on any plan that was previous mutated, we perform another step of mutation by recombining two districts in the plan. The concept of recombination (ReCom), first introduced by DeFord et al., is a way to perturb a district graph by combining two districts together

Algorithm 9 District Building

```
Choose $d$ clusters to be the centers for district creation:
 1: Choose five sets of $d$ clusters randomly. Each set is a candidate to become the set of centers.
 2: For each set of clusters, calculate the total graph distance between all pairs of centers. That is,
    for each set $i$, find $\mathcal{L}_i = \sum_{\substack{j=1 \\ j>k}}^{d} \sum_{k=1}^{d} \ell_{jk}^i$ where $\ell_{jk}^i$ is the graph distance between cluster $j$ and cluster
    $k$ in the cluster graph $CG_i$.
 3: Pick the set of set of centers that are maximally distant from one another to be the centers.
    That is, choose the set of clusters in set $\arg\max_i \mathcal{L}_i$.
Choose a district to grow and append a cluster to it:
 1: while Any cluster is unassigned do
 2:     Assign each district a weight that is inversely proportional to the current population of the
        district.
 3:     Based on these weights, use a weighted random selection to designate the district $\mathcal{D}$ that
        grows.
 4:     Assign a new weight to each unassigned neighboring cluster of $\mathcal{D}$ that is proportional to the
        boundary length with district $\mathcal{D}$.
 5:     Use a weighted random selection to designate the cluster that is appended to district $\mathcal{D}$. Add
        the cluster's population to the population for district $\mathcal{D}$, $p_{\mathcal{D}}$.
 6:     If $p_{\mathcal{D}} \geq 2\mathcal{I} * \alpha^{gen}$, then remove district $\mathcal{D}$ from the list of districts that are eligible to grow.
 7:     If it is found that no unassigned clusters exist around district $\mathcal{D}$, remove $\mathcal{D}$ from the list of
        districts that are eligible to grow.
 8: end while
 9: If this approach failed to assign every cluster to a district or if any district had population less
    than $0.15\mathcal{I}$, restart this procedure by choosing five new sets of centers.
10: Calculate all objective values for the plan.
11: If any objective exceeds the upper bound defined by the user, reject the plan and restart the
    procedure by choosing five new sets of centers.
```

and then redistributing the nodes of this graph into two new districts [9]. Our algorithm utilizes ReCom in the following way:

1. Find two neighboring districts, where one has population above ideal population and the other has population below ideal population.
2. Perform ReCom on these two districts using clusters as the nodes, ensuring that the resultant assignment of districts forces both districts to be within 5% of the mean population of the two districts.

This procedure helps to decrease the population deviation of the plan.

6.2.2.5 Nondominated Sorting and Crowding Distance

The remaining steps in our algorithm follow standard NSGA-II procedure closely. After district plans are created for the n children, we then decide which n of the $2n$ candidate plans are brought to the next generation. Let f_{ij} correspond to the ith objective value from the jth plan. We say that plan s_{j_1} *dominates* plan s_{j_2} if $f_{ij_1} \leq f_{ij_2}$ for all objectives i, and that for at least one objective i^*, we have $f_{i^*j_1} < f_{i^*j_2}$. We say that a plan s is *nondominated* if there is no other plan that dominates it.

Utilizing some subset of the objectives mentioned in section 6.1, we rank the $2n$ plans into fronts based on their domination status. All of the nondominated plans are placed in rank 1; all plans that are dominated *only* by plans in rank 1 are then placed in rank 2, and so on until all plans have a rank.

Then the best n plans (that is, the plans with the best rank) are designated to survive to the next generation. If a front must be split up to determine which plans survive, then we choose the plans that are the most diverse. To accomplish this, we calculate the *crowding distance* for all plans in the front that must be split up, as shown in Algorithm 10. Then, if x plans must be chosen from front F, we choose the x plans with the highest crowding distance to survive to the next generation. This ensures that the population is as diverse as possible.

Once the n plans are designated to survive to the next generation, we maintain their clustering assignments and their district assignments. When breeding children, the parents' clustering assignments are used in the crossover step, but the parents' district assignments are not considered when breeding children, as explained in subsection 6.2.2.2.

Algorithm 10 Crowding Distance

```
1: Initialize $cd_j = 0$ for each plan $j$ in front $F$
2: for each objective $i$ do
3:     Sort the $j$ plans in ascending order based on objective $i$
4:     $cd_0, cd_{|F|} \leftarrow \infty$
5:     $r \leftarrow f_{i|F|} - f_{i0}$                ▷ This is the range of objective $i$
6:     if $r = 0$ then
7:         $r \leftarrow 1$
8:     end if
9:     for $j = 1 \rightarrow |F|–1$ do
10:        $cd_j \leftarrow cd_j + \frac{f_{i(j+1)} - f_{i(j-1)}}{r}$
11:    end for
12: end for
```

This overall algorithm repeats until we have reached the predetermined number of generations.

6.3 Results

6.3.1 Experimental Design

To assess the efficacy of this program, we run both our code and the VNSGA-II using the same input parameters. Eight experiments are conducted with various objectives being evaluated in each experiment. Thirty independent trials of each experiment are performed. The summary of experiments can be found in Table 6.1. For each objective (Population Deviation, Polsby-Popper Score, Efficiency Gap, Median-Mean, and County Splits), an X is placed in the box if the objective is considered in the experiment. We choose this set of experiments because Population Deviation and Polsby-Popper Compactness are generally regarded as essential objectives when constructing a districting plan, while the other objectives may be de-emphasized or ignored entirely.

All experiments run for 200 generations. The mutation probability is set to 0.3 for all experiments. Experiments with more objectives require a larger population size n to fully approximate the Pareto front, so we utilize the following population sizes:

- Two objectives: $n = 25$
- Three objectives: $n = 75$
- Four objectives: $n = 150$

- Five objectives: $n = 250$

Each objective has an upper bound to ensure that absurd plans aren't being considered. The upper bounds used in this experiment are:

- Population Deviation: $PD \leq 0.4\mathcal{P}$, where $\mathcal{P}$ is the population of the state.
- Polsby-Popper Compactness: $PP_s \leq 0.9 \quad (\implies PP_i \leq 9)$.
- Efficiency Gap: $EG \leq 0.24$. This is three times the author-recommended EG limit of 0.08 [32].
- Median-Mean: $MM \leq 0.05$.
- County Splits: $CS \leq 50$.

Experiment Number	PD	PP	EG	MM	CS
1	X	X			
2	X	X	X		
3	X	X		X	
4	X	X			X
5	X	X	X	X	
6	X	X	X		X
7	X	X		X	X
8	X	X	X	X	X

Table 6.1: Summary of Experiments

For each generation, all objectives are scaled by their respective upper bounds so that they have the same relative impact on the hypervolume. The reference point is then placed at the coordinate $(1.1, 1.1, \ldots)$. As long as the entire approximate Pareto Front has cardinality less than or equal to the population size n, then we should expect monotonic increase from the hypervolume calculation.

We also compare our results to the actual 2024 South Carolina congressional plan, pictured in Figure 6.4.

6.3.2 Experimental Results

Table 6.3 reports the hypervolume graph summary for each experiment. Each experiment utilizes 30 trials; the dark line represents the median hypervolume in each generation for those thirty trials. The dark band surrounding the median represents the middle 50% of those hypervolume values. The light band extends as far as the farthest data point in $\left[\ell - \frac{3}{2} IQR, u + \frac{3}{2} IQR\right]$, where ℓ is the 25th percentile value, u is the 75th percentile value, and IQR is the interquartile range. Finally, red dots are placed for any outlier values outside the interval $\left[\ell - \frac{3}{2} IQR, u + \frac{3}{2} IQR\right]$.

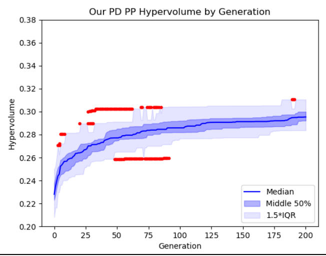
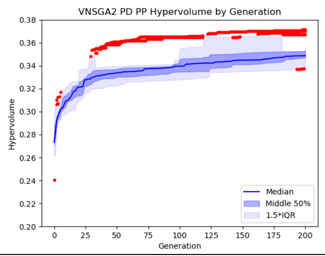
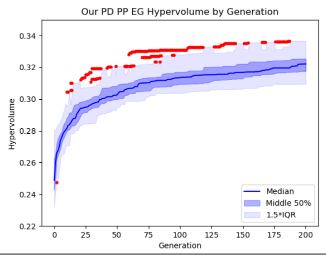
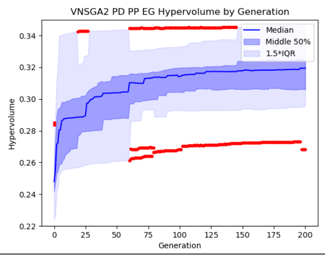
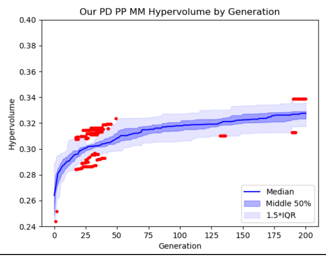
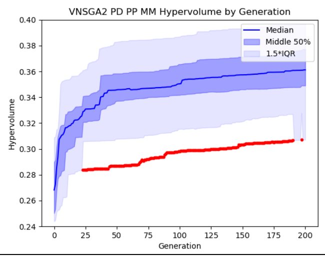
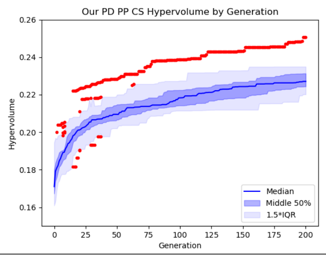
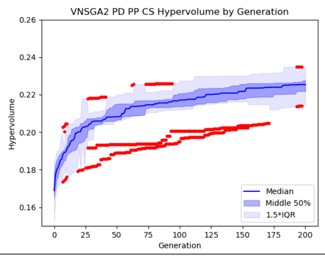
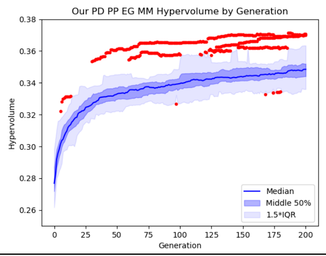
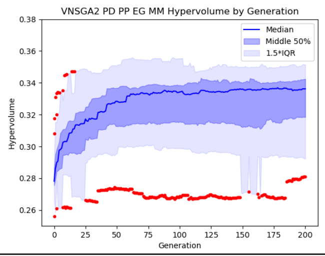
Exp	Objectives		Our Approach	VNSGA-II
1	PD:	X		
	PP:	X		
	EG:			
	MM:			
	CS:			
2	PD:	X		
	PP:	X		
	EG:	X		
	MM:			
	CS:			
3	PD:	X		
	PP:	X		
	EG:			
	MM:	X		
	CS:			
4	PD:	X		
	PP:	X		
	EG:			
	MM:			
	CS:	X		
5	PD:	X		
	PP:	X		
	EG:	X		
	MM:	X		
	CS:			

Table 6.3 continued from previous page

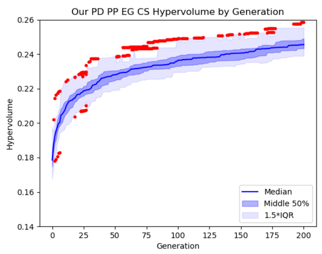
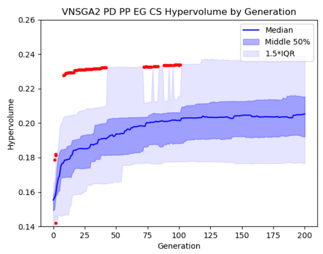
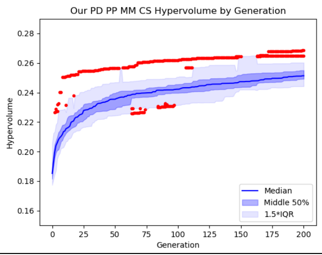
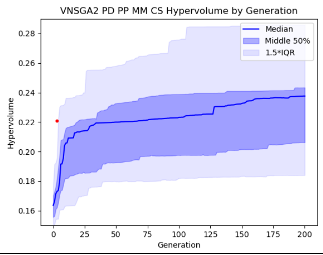
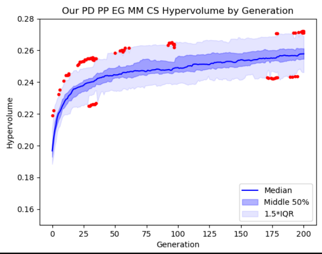
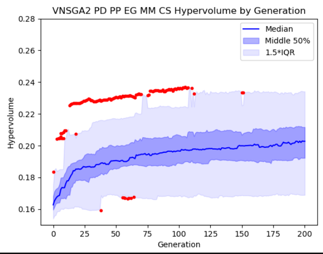
Exp	Objectives	Our Approach	VNSGA-II
6	PD: X		
	PP: X		
	EG: X		
	MM:		
	CS: X		
7	PD: X		
	PP: X		
	EG:		
	MM: X		
	CS: X		
8	PD: X		
	PP: X		
	EG: X		
	MM: X		
	CS: X		

Table 6.3: Hypervolume Graphs for Our Approach and VNSGA-II

This table provides evidence that our approach is competitive with VNSGA-II in most experiments. The hypervolume of our approach grows to larger values more quickly and more reliably than VNSGA-II in all experiments except 1 and 3. One observation to make is that the hypervolume for any experiment with four or more objectives is not monotonically increasing. This suggests that the population size n was not large enough for these experiments.

We also found that our approach completed quicker than the alternative. Table 6.4 reports the average runtime for each experiment. These experiments were coded in Python 3, and the experiments are performed on Clemson University's supercomputer using ten cores, an Intel Xeon chip, and 120 GB of memory. Profiling our rendition of Vanneschi's approach reveals that the variable neighborhood search expended the most time while running. We leave open the possibility that this code could be run more efficiently with better implementation. Our code is documented in Appendix D: NSGA-II Codes, and our rendition of Vanneschi's code is found in Appendix E: VNSGA-II Codes.

	Our Runtimes (s)				VNSGA-II Runtimes (s)			
Exp	Avg	St. Dev.	Min	Max	Avg	St. Dev.	Min	Max
1	5757	150	5550	6242	31336	4170	18646	39189
2	17195	173	16897	17606	57528	6027	50422	71822
3	15223	751	14347	16546	68035	5971	57800	80189
4	15429	212	15071	16011	58672	5832	55071	71239
5	32658	2142	30903	39188	150388	10249	127630	169536
6	35450	4029	31740	41796	135301	10154	121974	159130
7	33287	1923	31167	38776	131429	14928	115112	178401
8	55934	3068	48687	58255	228923	13170	205177	253331

Table 6.4: Average Runtimes for each Experiment in Seconds

Table 6.5 reports the average size of the final Pareto front for each experiment. A large final front is neither good nor bad, but it does help to enlighten a decision maker on the number of plans that would need to be chosen from. This table demonstrates that it might be prudent to increase the population size to fit experiments with more objectives.

	Our PF Sizes			VNSGA-II PF Sizes		
Exp	Avg	Min	Max	Avg	Min	Max
1	29	18	40	19	15	28
2	65	51	76	105	90	116
3	52	26	68	93	56	107
4	27	17	40	101	72	130
5	155	139	168	218	202	240
6	145	105	154	206	188	219
7	123	79	151	194	179	236
8	264	241	286	353	320	383

Table 6.5: Average Pareto Front (PF) Sizes at the Conclusion of each Experiment

We now display a high quality plan produced by our approach and compare it to the 2024 implemented congressional plan. This high-quality plan found by our algorithm is displayed in Figure 6.4. This plan has incredibly small objectives relative to its peers in the same experiment and improves upon the current congressional map in every objective except for county splits.

Plan	*PD*	PP_s	*EG*	*MM*	*CS*
2024 Actual	14136	0.785	0.248	0.032	10
Exp 15, Trial 23, Plan 214	12615	0.784	0.101	0.024	23

Table 6.6: Comparison of Objectives between Current Congressional Plan and our Plan

We also record the Mean Ideal Gap metrics for both our approach (Figure 6.5) and VNSGA-

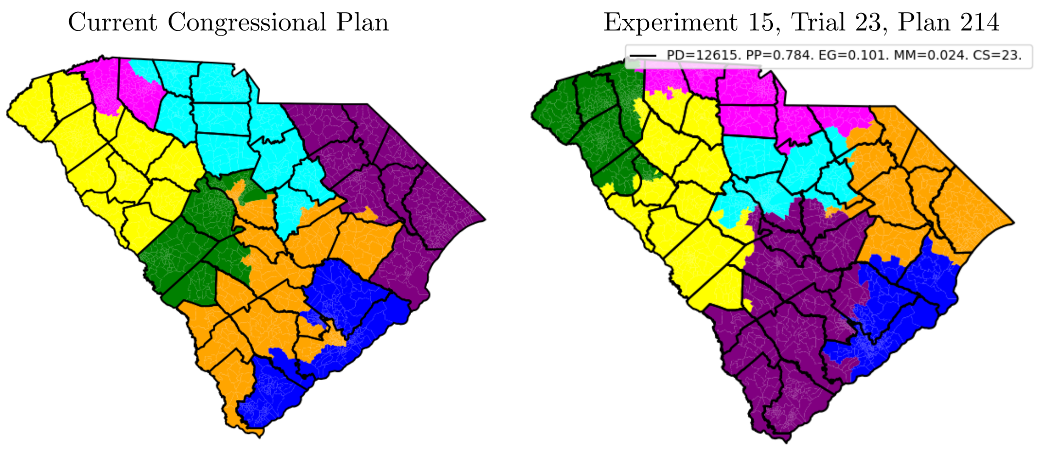

Current Congressional Plan
Experiment 15, Trial 23, Plan 214
PD=12615. PP=0.784. EG=0.101. MM=0.024. CS=23.

Figure 6.4: Comparison of Plans

II (Figure 6.6). As mentioned in section 5.1, smaller MIG values are better. We observe that as the number of objectives increases, the value of the MIG also increases, demonstrating worse performance.

Finally, we report the covered size of space metric (CSS) mentioned in section 5.1. This metric compares two sets of Pareto fronts by determining the percentage of the first front that is dominated by the second. This gives some insight into how well the algorithm actually performed. To assess the efficacy of our algorithm, we compile all trials of a particular experiment into a single nondominated Pareto set P_N for our code and compare it to the compiled nondominated Pareto set for VNSGA-II P_V. Thus, the quantity $C(P_N, P_V)$ reports the percentage of plans in P_V that are dominated by any number of plans in P_N. In Table 6.7, we report both $C(P_N, P_V)$ and $C(P_V, P_N)$, as $1 - C(P_N, P_V)$ is not necessarily equal to $C(P_V, P_N)$.

Exp	$C(P_N, P_V)$	$C(P_V, P_N)$
1	0	1
2	0.3440	0.3866
3	0	0.9655
4	0.5469	0.0811
5	0.1775	0.3704
6	0.3325	0.1671
7	0.1277	0.6131
8	0.6038	0.0727

Table 6.7: The Covered Size of Space Metric Values Comparing Our Approach to VNSGA-II

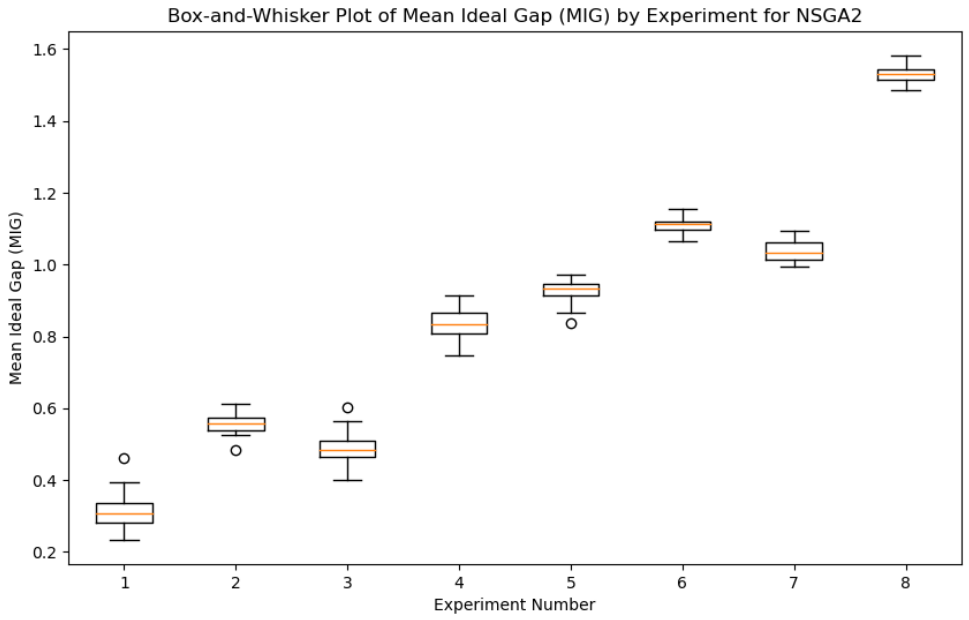


Figure 6.5: Mean Ideal Gap for Our Approach

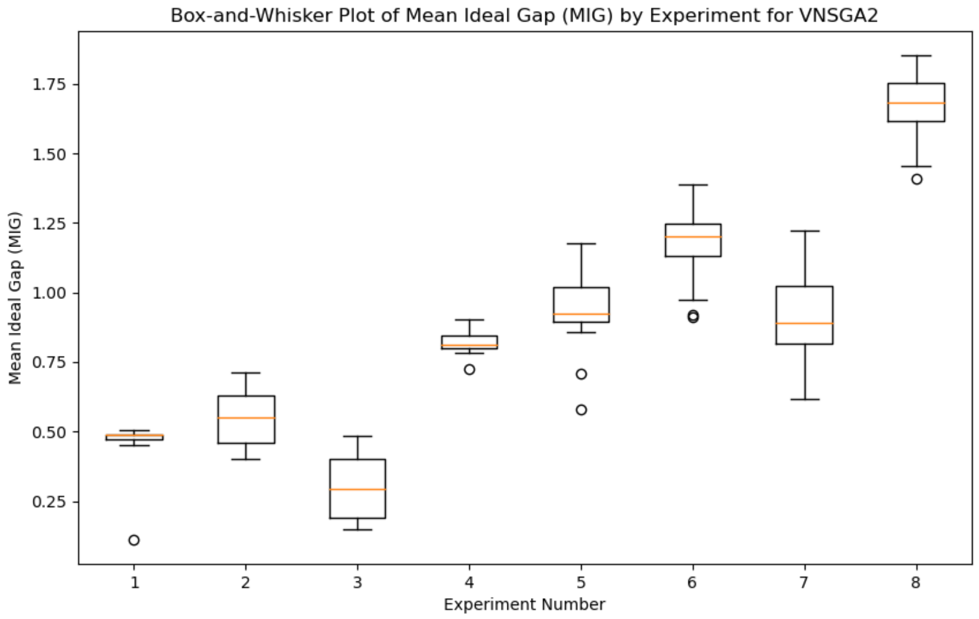


Figure 6.6: Mean Ideal Gap for VNSGA-II

6.4 Conclusion and Future Work

Overall, we conclude that utilizing genetic algorithms and in particular, NSGA-II has promise for optimizing various objectives when making districting plans. We show that utilizing a novel clustering technique can be an effective tool for building districts, and that it follows the theme of genetic algorithms closely. Despite being outperformed on the two-objective experiment, we find that our algorithm works well on nearly every other experiment. We demonstrate that this algorithm is competitive in both computation time and results with another similar NSGA-II algorithm developed by Vanneschi et al.

Future work for this project may include the following improvements or changes:

- It remains to be seen whether the objectives could be lowered further with more generations. Due to time constraints, this code needed to terminate at generation 200, but the overall trend of the hypervolume graphs suggests that local optima were not yet reached.

- Other objectives, such as similarity to previous plans, grouping communities of interest, and adherence to the Voting Rights Act, could still be included and measured as part of a multi-objective framework.

- We used South Carolina as a test case in these examples; future work could include testing on larger states with more districts.

- The CSS metric in Table 6.7 seems to imply that more work should be done to improve the coverage of our algorithm. One such approach might be to have children inherit some data about the district assignment, such as the location of the district center.

Chapter 7

Conclusions

In conclusion, this dissertation demonstrates four unique approaches for optimizing the redistricting process without human bias. Chapter 3 demonstrates that trying to find optimal solutions even for a single objective can be very challenging with so many constraints and variables. Chapters 4, 5, and 6 all show approaches to optimize one or more objectives. Results from Chapter 4 demonstrate an effective modification to simulated annealing that minimizes inverse Polsby-Popper compactness. We find a score of $PP_i = 1.65$, while maintaining population deviation within 1% of ideal. This effectively demonstrates that the implemented Congressional maps sacrificed compactness for the sake of achieving some other aim.

Necessarily, the multiobjective algorithms in Chapters 5 and 6 perform worse on the compactness measures for the sake of also achieving minimization of other objectives. The plans achieved in these chapters still should be considered high-quality outputs as they sit on an approximate Pareto front and achieve remarkably small efficiency gap and median-mean objectives.

Appendices

Appendix A MIP Codes

```
from gerrychain import Graph
import geopandas as gpd
import networkx as nx
import gurobipy as gp
from gurobipy import GRB
import pickle
import pandas as pd
import matplotlib.pyplot as plt
import sys
import os



class input_c: #short for 'input class'
    def __init__(self, nd, gu, lb, ub, obj, psf, ppn, tl):
        self.num_dists = nd
        self.GU = gu
        self.pop_lb = lb
        self.pop_ub = ub
        self.obj = obj
        if psf == "NoStart": pass
        elif os.path.exists(psf) == False: raise FileNotFoundError(f"File
           {psf} not found")
        self.ps_file = psf #Past solution file
        self.ps_plan_no = ppn #Past solution plan number
        self.timelimit = tl

    @classmethod
    def Load_from_file(cls, file_name):
        """Reads the input from a file"""
        with open(file_name) as f:
            lines = f.readlines()
        num_dists=int(lines[0])
        geo_units=lines[1].strip()
        lower_bound=float(lines[2])
        upper_bound=float(lines[3])
        objective=lines[4].strip()
        pastsoln_file=lines[5].strip()
        ps_plan_no = int(lines[6])
        timelimit = int(lines[7])
        return cls(
            nd=num_dists,
            gu=geo_units,
            lb=lower_bound,
            ub=upper_bound,
            obj=objective,
            psf=pastsoln_file,
            ppn = ps_plan_no,
            tl=timelimit
        )
```

```
def plot_plan(G, county_lines, ip, status, objval):
    """Plots one plan with county lines"""
    dists = list(dict(G.nodes("dist")).values())
    G.data["dist"] = dists
    fig, ax = plt.subplots()
    for idx, gu in G.data.iterrows():
        if ip.GU == "counties":
            ax.text(gu.geometry.centroid.x, gu.geometry.centroid.y, idx,
                fontsize=8)
    G.data.plot(column="dist", ax=ax, legend=True)
    county_lines.data.boundary.plot(ax=ax, color="black")
    if status == 2: title = f"Optimal {ip.obj} Plan. {ip.obj}={objval}"
    elif status == 9: title = f"TimeLimit {ip.obj} Plan. {ip.obj}={objval}"
    elif status == 15: title = f"Cutoff {ip.obj} Plan. {ip.obj}={objval}"
    legend_label = f"D={ip.num_dists}, LU={ip.pop_lb}-{ip.pop_ub}"
    ax.set_title(title)
    ax.legend([legend_label])
    fig.show()



def main(*args):

    #I'm now using machine-specific license to do this.
    # options = { #For accessing Gurobi license
    #     #"WLSACCESSID": "efcde7b0-5d6f-477b-ab3a-d675f56c5daa",
    #     #"WLSSECRET": "befb233b-609f-42df-a823-d5937d6253b7",
    #     "LICENSEID": 2459526,
    # }

    #INPUTS------------------------------------------------------------
    if isinstance(args[0], str) == True:
        file_name = args[0]
        ip = input_c.Load_from_file(file_name)
    elif args[0] == None:
        print("No input file provided. Running with default parameters")
        ip = input_c(
            nd=3,
            gu="counties",
            lb=0.8,
            ub=1.2,
            obj="Compactness",
            psf=None,
            ppn=None
        )



    #ERROR CHECKS------------------------------------------------------
    if (ip.num_dists%2==0 and ip.obj == "MM"):
        raise ValueError("This median-mean formulation only accepts odd
            numbers of districts. Use 'MM2'. ")
    if ip.GU == "counties" and ip.obj == "CS":
        raise ValueError("The county splits objective must use precincts as
            a GU")
```

```
with open('stateG.pkl', 'rb') as fp: stateG = pickle.load(fp)
with open('county_boundaries.pkl', 'rb') as fp: countyG =
    pickle.load(fp)
# try:
#     with open('county_boundaries.pkl', 'rb') as fp: countyG =
    pickle.load(fp)
# except AttributeError:
#     countyG = Graph.from_file(
#         "./SC_Counties_20_FeaturesToJSO.geojson",
#         adjacency="rook",
#         reproject="True",
#         ignore_errors="True"
#     )
#     with open('county_boundaries.pkl', 'wb') as fp:
    pickle.dump(countyG, fp)


#PARAMETERS AND SETS----------------------------------------------------
with gp.Env() as env, gp.Model(env=env) as m:
    m.params.NonConvex=2
    m.params.TimeLimit=ip.timelimit #255600 seconds = 71 hours

    if ip.obj == "Compactness":
        m.params.Cutoff= float('inf')
    #m.params.FeasibilityPump = 1 --- Not sure that this is even a thing
    #m.params.PumpPasses = 10 --- Doesn't work since there are
        quadratic constraints in the model
    if ip.GU == "precincts":
        V = stateG.number_of_nodes()
        E = stateG.number_of_edges()
        G = stateG
    elif ip.GU == "counties":
        V = countyG.number_of_nodes()
        E = countyG.number_of_edges()
        G = countyG
    else:
        raise ValueError("Incorrect GU type")

    D = ip.num_dists
    C = len(set(dict(stateG.nodes("COUNTY")).values())) #Number of
        counties
    fwd_E_list = list(G.edges())
    backward_E_list = [(v2, v1) for (v1,v2) in fwd_E_list]
    E_list = fwd_E_list + backward_E_list

    #DECISION VARIABLES--------------------------------------------
    x = m.addVars(V, D, vtype=GRB.BINARY, name="x") #decision variable
        for GU-district assignment
    f = m.addVars(E_list, D, vtype=GRB.CONTINUOUS, lb=0, name="f")
        #flow variables for contiguity
    w = m.addVars(V, D, vtype=GRB.BINARY, name="w") #w_ik = 1 if GU i
        is assigned to be the hub for district k
    pop = m.addVars(D, vtype=GRB.INTEGER, name="pop") #population
        variables for districts
```

```
popdev = m.addVars(D, vtype=GRB.INTEGER, lb=0, name="popdev")
    #Population deviation for each district

if ip.obj == "Compactness":
    P = m.addVars(D, vtype=GRB.CONTINUOUS, name="P") #perimeter of
        each district
    A = m.addVars(D, vtype=GRB.CONTINUOUS, name="A") #area of each
        district
    z = m.addVars(D, vtype=GRB.CONTINUOUS, lb=0, name="z") #inverse
        Polsby-Popper Score of each district
    y = m.addVars(E_list, D, vtype=GRB.BINARY, name="y") #decision
        variable. =1 if edge e is a district edge
elif ip.obj == "CS": #County splits
    t = m.addVars(C, D, vtype=GRB.BINARY, name="t") #t_cj = 1 if
        county c contains any part of district j
elif ip.obj == "EG": #Efficiency Gap
    b = m.addVars(D, vtype=GRB.INTEGER, lb=0, name="b") #blue votes
        in the jth district
    r = m.addVars(D, vtype=GRB.INTEGER, lb=0, name="r") #red votes
        in the jth district
    I = m.addVars(D, vtype=GRB.BINARY, name="I") #Indicator
        variable. =1 if b_j >= r_j, 0 o/w
    eg = m.addVar(vtype=GRB.CONTINUOUS, name="eg") #Variable used
        to account for absolute value around efficiency gap
    real_eg = m.addVar(vtype=GRB.CONTINUOUS, name="real_eg")
        #Variable that finds the real efficiency gap (eg is used
        for better computation)
elif ip.obj == "MM": #Median-Mean
    b = m.addVars(D, vtype=GRB.INTEGER, lb=0, name="b") #blue votes
        in the jth district
    r = m.addVars(D, vtype=GRB.INTEGER, lb=0, name="r") #red votes
        in the jth district
    med = m.addVar(vtype=GRB.CONTINUOUS, name="med") #median-mean
        calculation
    uu = m.addVars(D, vtype=GRB.BINARY, name="uu") #If uu_i=1, then
        med>=y_i
    vv = m.addVars(D, vtype=GRB.BINARY, name="vv") #If vv_i=1, then
        med<=y_i
    bsp = m.addVars(D, vtype=GRB.CONTINUOUS, lb=0, name="bsp")
        #Blue share for president (=b/(b+r))
    amm = m.addVar(vtype=GRB.CONTINUOUS, name="amm") #absolute
        value of median mean
elif ip.obj == "MM2": #Median-Mean (second formulation)
    b = m.addVars(D, vtype=GRB.INTEGER, lb=0, name="b") #blue votes
        in the jth district
    r = m.addVars(D, vtype=GRB.INTEGER, lb=0, name="r") #red votes
        in the jth district
    med = m.addVar(vtype=GRB.CONTINUOUS, name="med") #median-mean
        calculation
    bsp = m.addVars(D, vtype=GRB.CONTINUOUS, lb=0, name="bsp")
        #Blue share for president (=b/(b+r))
    so = m.addVars(D, vtype=GRB.CONTINUOUS, lb=0, name="so")
        #Sorted bsp's
    p = m.addVars(D,D, vtype=GRB.BINARY, name="p") #permutation
        matrix that sorts the bsp's
    amm = m.addVar(vtype=GRB.CONTINUOUS, name="amm") #absolute
        value of median mean
elif ip.obj == "PopDev": #Population Deviation
    m.params.BestObjStop=0
m.update()
print("Decision variables loaded")


#START
    VALUES-----------------------------------------------------------------
if ip.ps_file == "Default":
    #Establishes the past solution to be used as an incumbent if
        not specified
    fn = "IP_Red_{}d_{}_obj{}_{}-{}.xlsx".format(ip.num_dists,
        ip.GU[0], ip.obj, round(100*ip.pop_lb),
        round(100*ip.pop_ub))
    if os.path.exists(fn): #else no file provides starting values
        ip.ps_file = fn
        psdf = pd.read_excel(ip.ps_file, sheet_name="Results")
            #psdf = past solution data frame
        for _, row in psdf.iterrows(): #Assumes data is arranged as
            [Variable, index, value]
            var_name = row["Variable"]
            index_str = row["index"]
            value = row["value"]

            if pd.isna(index_str):
                var = m.getVarByName(var_name)
            else:
                index = index_str.replace(" ", "")  # Remove the
                    space after commas
                var = m.getVarByName(f"{var_name}{index}")

            # Set start value
            if var is not None:
                var.start = value
        print(f"Starting variables loaded from {ip.ps_file}")
    else: print("No starting variables loaded")

elif ip.ps_file[:5] == "NSGA2": #if the starting values come from
    an NSGA2 output
    psdf = pd.read_excel(ip.ps_file, sheet_name="GU Assignment")
    final_gen = max(psdf["Generation"])
    filtered_df = psdf[(psdf['Generation'] == final_gen) &
        (psdf['pop_id'] == ip.ps_plan_no)]
    for _, row in filtered_df.iterrows():
        node = row["node"] - 1
        dist = row["district"]
        x[node, dist].start = 1
    print(f"Starting variables loaded from {ip.ps_file}")
elif ip.ps_file[:6] == "IP_Red": #the starting values come from
    IP_Red output
    psdf = pd.read_excel(ip.ps_file, sheet_name="Results")
    for _, row in psdf.iterrows(): #Assumes data is arranged as
        [Variable, index, value]
        var_name = row["Variable"]
        index_str = row["index"]
        value = row["value"]

        if pd.isna(index_str):
            var = m.getVarByName(var_name)
        else:
            index = index_str.replace(" ", "")  # Remove the space
                after commas
            var = m.getVarByName(f"{var_name}{index}")

        # Set start value
        if var is not None and var_name=='x':
            var.start = value
    print(f"Starting variables loaded from {ip.ps_file}")
elif ip.ps_file == "NoStart":
    print("No starting variables loaded")

try: del psdf
except NameError: pass
try: del filtered_df
except NameError: pass
m.update()



#EXTRA CALCULATIONS---------------------------------------
edge2num = {} #dictionary for enumerating the edges. e.g.,
    edge2num[(0,1)] = 0
idx=0
for edge in sorted(G.edges()):
    edge2num[edge] = idx
    idx=idx+1

total_votes = sum(dict(G.nodes("PresBlue")).values()) + \
              sum(dict(G.nodes("PresRed")).values())

#Adjacency matrix
adj = {}
for u in range(V):
    for v in range(V):
        adj[(u,v)] = 0 #sets baseline of 0
        adj[(v,u)] = 0
for (u,v) in G.edges():
    adj[(u,v)] = 1
    adj[(v,u)] = 1

#County numbers
if ip.GU == "precincts":
    county_list = set()
    for n in G.nodes:
        county_list.add(G.nodes[n]['COUNTY'])
    county_list = sorted(county_list)
    # Long county name to short county name. e.g.
        longc2shortc['45003'] = 1
    longc2shortc = {}
    for idx, c in enumerate(county_list):
        longc2shortc[c] = idx
    for n in G.nodes:
        G.nodes[n]['county_num'] =
            longc2shortc[G.nodes[n]['COUNTY']]

#GU-county relationship
if ip.GU == "precincts":
    s = {}
    for i in range(V):
        for c in range(C):
            if G.nodes[i]['county_num'] == c:
                s[(i,c)] = 1 #s[i,c] = 1 if GU i is in county c
            else:
                s[(i,c)] = 0

#Ideal Population
ideal_pop = sum(dict(G.nodes("POPULATION")).values()) / D
print("Extra Calculations finished")


#OBJECTIVE----------------------------------------------------
if ip.obj == "Compactness":
    m.setObjective(1/D * sum(z[j] for j in range(D)), GRB.MINIMIZE)
        #inverse polsby-popper
elif ip.obj == "CS":
    m.setObjective(sum(sum(t[c,j] for j in range(D)) for c in
        range(C)), GRB.MINIMIZE) #county splits
elif ip.obj == "EG":
    m.setObjective(real_eg, GRB.MINIMIZE)
elif ip.obj == "MM":
    m.setObjective(amm, GRB.MINIMIZE)
elif ip.obj == "MM2":
    m.setObjective(amm, GRB.MINIMIZE)
elif ip.obj == "PopDev":
    m.setObjective(sum(popdev[j] for j in range(D)), GRB.MINIMIZE)
else: raise NameError("ip.obj is not input correctly. ip.obj =
    {}".format(ip.obj))

m.addConstrs(sum(x[i,j] for j in range(D)) == 1 for i in range(V))
    #Each GU assigned to one dist
print(f"Objective loaded: {ip.obj}")

#CONSTRAINTS--------------------------------------------------
#Compactness constraints
if ip.obj == "Compactness":
    m.addConstrs(4*3.14159265*A[j]*z[j] >= P[j]*P[j] for j in
        range(D))
    m.addConstrs(A[j] == sum(G.nodes[i]["area"]*x[i,j] for i in
        range(V)) for j in range(D))
    m.addConstrs(P[j] == (sum(G.nodes[i]["boundary_perim"]*x[i,j]
        for i in range(V)) +
                          sum(G.edges[(u,v)]['shared_perim']*y[u,v,j]
                              for (u,v) in G.edges())) for j in
```

```
                         range(D))
        m.addConstrs(x[u,j] - x[v,j] <= y[u,v,j] for (u,v) in G.edges()
            for j in range(D)) #Cut edges
        m.addConstrs(x[v,j] - x[u,j] <= y[u,v,j] for (u,v) in G.edges()
            for j in range(D)) #Cut edges

    #County Splits constraints
    elif ip.obj == "CS":
        m.addConstrs(sum(x[i,j]*s[(i,c)] for i in range(V))/V <= t[c,j]
            for c in range(C) for j in range(D))

    #Efficiency Gap constraints
    elif ip.obj == "EG":
        m.addConstrs(b[j] == sum(G.nodes[i]["PresBlue"]*x[i,j] for i in
            range(V)) for j in range(D)) #defines blue votes per dist
        m.addConstrs(r[j] == sum(G.nodes[i]["PresRed"]*x[i,j] for i in
            range(V)) for j in range(D)) #defines red votes per dist
        m.addConstr(eg >= sum((b[j]-3*r[j]-1)*I[j] +
            (3*b[j]-r[j]+1)*(1-I[j]) for j in range(D)))
        m.addConstr(eg >= -sum((b[j]-3*r[j]-1)*I[j] +
            (3*b[j]-r[j]+1)*(1-I[j]) for j in range(D)))
        m.addConstrs(I[j] <= 1+ (b[j]-r[j])/total_votes for j in
            range(D))
        m.addConstrs(I[j] >= (b[j]-r[j])/total_votes for j in range(D))
        m.addConstr(real_eg == eg/(2*total_votes))
    elif ip.obj == "MM":
        m.addConstrs(b[j] == sum(G.nodes[i]["PresBlue"]*x[i,j] for i in
            range(V)) for j in range(D)) #defines blue votes per dist
        m.addConstrs(r[j] == sum(G.nodes[i]["PresRed"]*x[i,j] for i in
            range(V)) for j in range(D)) #defines red votes per dist
        m.addConstrs((b[j]+r[j])*bsp[j] == b[j] for j in range(D))
            #Blue share for president
        k = (D-1)/2 #half of the number of districts rounded down
        m.addConstr(sum(uu[j] for j in range(D)) == k+1) #Ensures that
            only k+1 uu[j]'s are 1
        m.addConstr(sum(vv[j] for j in range(D)) == k+1) #Ensures that
            only k+1 vv[j]'s are 1
        M = 100000 #A sufficiently large number
        m.addConstrs(1000*bsp[j] - 1000*med <= M*(1-uu[j]) for j in
            range(D)) #If uu[j]=1, then bsp[j] <= med
        m.addConstrs(1000*med - 1000*bsp[j] <= M*(1-vv[j]) for j in
            range(D)) #If vv[j]=1, then bsp[j] >= med
        m.addConstr(1000*amm >= 1000*med - 1000/D * sum(bsp[j] for j in
            range(D))) #Ensures that the absolute value of mm is
            positive
        m.addConstr(1000*amm >= 1000/D * sum(bsp[j] for j in range(D))
            - 1000*med) #The 1000 is present so that rounding errors
            are less likely
    elif ip.obj == "MM2":
        m.addConstrs(b[j] == sum(G.nodes[i]["PresBlue"]*x[i,j] for i in
            range(V)) for j in range(D)) #defines blue votes per dist
        m.addConstrs(r[j] == sum(G.nodes[i]["PresRed"]*x[i,j] for i in
            range(V)) for j in range(D)) #defines red votes per dist
        m.addConstrs((b[j]+r[j])*bsp[j] == b[j] for j in range(D))
            #Blue share for president
        m.addConstrs(so[k] == sum(p[j,k]*bsp[j] for j in range(D)) for
            k in range(D)) #s[k] are the sorted bsp values
        m.addConstrs(so[k+1] >= so[k] for k in range(D-1)) #so[0] is
            smallest; s[D-1] is largest
        m.addConstrs(sum(p[j,k] for j in range(D)) == 1 for k in
            range(D))
        m.addConstrs(sum(p[j,k] for k in range(D)) == 1 for j in
            range(D)) #this and previous define a permutation matrix of
            size D*D
        if D%2 == 1: #Odd
            m.addConstr(med == so[(D-1)/2]) #Middle value
        elif D%2 == 0: #Even
            m.addConstr(med == (so[(D-2)/2] + so[D/2])/2) #mean of two
                middle values
        m.addConstr(1000*amm >= 1000*med - 1000/D * sum(bsp[j] for j in
            range(D))) #Ensures that the absolute value of mm is
            positive
        m.addConstr(1000*amm >= 1000/D * sum(bsp[j] for j in range(D))
            - 1000*med) #The 1000 is present so that rounding errors
            are less likely
    #elif ip.obj == "PopDev": # I want to know popdev even if using
        other metrics
    m.addConstrs(popdev[j] >= pop[j] - ideal_pop for j in range(D))
    m.addConstrs(popdev[j] >= ideal_pop - pop[j] for j in range(D))

    #Population equinumerosity
    m.addConstrs((ip.pop_lb*ideal_pop <= pop[j] for j in range(D)),
        name="pop_lb")
    m.addConstrs((ip.pop_ub*ideal_pop >= pop[j] for j in range(D)),
        name="pop_ub")
    m.addConstrs((pop[j] == sum(G.nodes[i]["POPULATION"]*x[i,j] for i
        in range(V)) for j in range(D)), name="pop_lb")

    #Contiguity
    m.addConstrs(sum(w[i,j] for i in range(V)) == 1 for j in range(D))
    m.addConstrs(sum(f[u,v,j] for v in range(V) if (u,v) in G.edges())
        - sum(f[v,u,j] for v in range(V) if (u,v) in G.edges()) >=
        x[u,j] - V*w[u,j] for u in range(V) for j in range(D))
    m.addConstrs(sum(f[v,u,j] for v in range(V) if (u,v) in G.edges())
        <= (V-1)*x[u,j] for j in range(D) for u in range(V))

    print("Constraints loaded")

    #OPTIMIZE--------------------------------------------------------
    m.optimize()

    #PRINT RESULTS---------------------------------------------------
    feas = True
    if m.status in [2,9,15]: #If optimal solution found, time limit
        reached, or cutoff found
        if ip.GU == "counties":
            for vr in m.getVars():
                try:
                    if abs(vr.X) > 0.00001: #Only print nonzero
                        variables
```

```
                            print('%s = %g' % (vr.VarName, vr.X))
                    except AttributeError: #This occurs if time limit is
                        reached and no feasible solution has been found.
                        feas = False #variable for whether a feasible
                            solution was found
                        print("Time limit reached and no feasible solution
                            was found.")
                        break

            for (i, j), vr in x.items(): #gu = i, dist = j
                if feas == True:
                    if vr.X > 0.9:  # best to allow for numerical tolerances
                        G.nodes[i]["dist"] = j
                else: break #This occurs if time limit is reached and no
                    feasible solution has been found.

            ip.runtime=m.Runtime
            ip.status=m.status
            ip.objval=m.ObjVal

            #SAVING TO EXCEL-------------------------------------------
            #fields = ["num_dists", "GU", "pop_lb", "pop_ub", "obj"]
            input_data = [list(ip.__dict__.values())]
            input_columns = list(ip.__dict__.keys())
            df0 = pd.DataFrame(input_data, columns=input_columns)

            excel_doc_name =
                "IP_Red_{}d_{}_obj{}_{}-{}.xlsx".format(ip.num_dists,
                ip.GU[0], ip.obj, round(100*ip.pop_lb),
                round(100*ip.pop_ub))
            if feas == True:
                df1 = pd.DataFrame({'Variable': var.VarName.split('[')[0],
                    'index':[int(i) for i in
                    var.VarName.split('[')[1][:-1].split(',')] if '[' in
                    var.VarName else None, 'value': var.X} for var in
                    m.getVars() if var.X>0.00001)
            else:
                df1 = pd.DataFrame()
            excel_writer = pd.ExcelWriter(excel_doc_name)

            df0.to_excel(excel_writer, sheet_name='Input')
            df1.to_excel(excel_writer, sheet_name='Results')
            excel_writer.save()
            print(f"Saved to {excel_doc_name}")

            #PLOTTING RESULTS------------------------------------------
            if feas: plot_plan(G, countyG, ip, m.status, m.ObjVal)

        print("done with model")

    print("finished")

if __name__ == "__main__":
    if len(sys.argv) == 1:
        main('IP_input_c.txt') #Use file name if reading from file
    elif len(sys.argv) == 2:
        input_file = sys.argv[1]
        main(input_file)
```

Appendix B Simulated Annealing Codes

The following Matlab program in Listing 1 is the main code used to conduct our experiment. We describe the main ideas behind the code here:

- Lines 20-39: We can change our input values here. Some inputs such as line 20 and 38 are broken, and will not work unless set to a specific number.
- Lines 83-129: We load our data.
- Line 165: Determines which precincts are on district boundaries.
- Line 273: We calculate the initial Polsby-Popper score for each district.
- Lines 282-287: We calculate the number of precincts in each CDI.
- Line 298: We start our main loop. This loop will iterate a maximum of 1,000,000 times and will swap exactly one precinct over district lines at each iteration.
- Lines 302-329: The various cooling schedules are defined.
- Lines 337-353: The exponent associated with the population constraint is defined based on the current iteration. The closer we are to the end of the algorithm, the larger this exponent becomes.
- Lines 390-427: A while loop that only loops more than once if all of the boundary precincts would break district contiguity if chosen.
- Line 392: We select n precincts on district boundaries.
- Lines 404-405: We calculate how each district's area, perimeter, Polsby-Popper score, and population would be affected if the ith precinct would be swapped across district lines.
- Lines 410-412: We compute the ΔE values for each of the n boundary precincts.
- Lines 413-414: We calculate the population scaling variable.
- Lines 425-426: Weights are assigned to the n precincts, and one is chosen to be swapped based on those weights.
- Lines 433-436, 454-473: Various values are updated after the precinct swap.

- Lines 441-451, 489-497: Results are recorded.
- Lines 506-519: If we use cooling schedule C, we test to see if we compactness is getting worse. If it is starting to level out, we switch to cooling.
- Lines 532-554: Determines if the algorithm has reached a local optimum. If the slope is shallow enough, we advance the iteration count to the next 10% benchmark.

Listing 1: Main Code

```
%% Input
clear
clc
close all

TrialNum = 7;
TS = num2str(TrialNum);
directoryname = ['Trial',TS];
mkdir(directoryname)

for run=1:1

MAXPREC = 2232; %Number of precincts
MAXDIST = 7; %Number of districts
MAXNHB = 21; %Maximum precinct neighbor count
MAXCOUNTY = 46; %Number of counties

%Choose 0 if compactness is preferred
%Choose 1 if political boundaries are preferred
input.pref=0; % OPTION 1 DOES NOT WORK
input.Exp=10; %The largest Exp is, the more we value pref chosen above.
input.Exp2=1; %0 or 1. 0 if we don't care about political boundaries
input.n=30; %Number of precincts sampled
input.Itmax=1; %Number of iterations
input.power=1; %Default power for population control. Will change later.
input.comments=0; %1 for command window statements, 0 otherwise
input.CS=2; %Cooling Schedule. 0 for Immediate drop; 1 for Geometric; 2 for
    multiple cooling rates
input.MCL =input.Itmax/1000; %Markov Chain Length. This is the number of
    iterations at each temperature
input.initT=100; %Initial Temperature--May be overridden depending on
    input.CS.
input.alpha = 0.985; %Cooling rate
input.figures =1; %0 for suppressing images; 1 for displaying B/M/E; 2 for
    more maps
input.BM =0; %0 for suppressing Boundary Maps; 1 for displaying them.
input.PPL =1; %Population Percent Limit. Must be in (0,5]. Restricts
    population limits for districts (i.e. 5 means each district is +-5% of
    targetpop)
input.saving=1; %Saving variable. Determines whether we want to save
    certain variables. 0 for no, 1 for yes.
```

```
input.video=0; %Tells us if we want to save a video. 0 for no, 1 for yes.
input.videospace = 5000; %Space between the video frames. Records a picture
    every input.videospace iterations
input.eps = .05/2000; %Epsilon. if slope of last 2000 compactness scores is
    between -eps and +eps, we advance the code.
input.optcomp=0; %Tells us whether to use the optcomp function or not.
    DOESN'T WORK
input.damping=1; %Tells us whether to use damping factor or divide all
    entries by largest. 0 for no damping factor

if run ==1
    currdate = datestr(clock,'mm-dd-yy_HH-MM');
end
runnum = num2str(run);

if input.saving ==1 && run ==1
    String = [directoryname,'/Input_',currdate,'.mat'];
    save(String,'input');
end

%% Initialization
if run==1
    fig =0; % Increases by 1 each time a new figure is needed
end
FinalMapIt=input.Itmax; %Default index value for final map.
T=-inf;
clear Results

%Initializes the Results struct
Results(input.Itmax).DistPop = zeros(MAXDIST,1);
Results(input.Itmax).PopDiff = zeros(MAXDIST,1);
Results(input.Itmax).PopScale = -inf;
Results(input.Itmax).DeltaE = -inf;
Results(input.Itmax).DeltaComp = -inf;
Results(input.Itmax).CompactnessScore = -inf;
Results(input.Itmax).MovedPrec = -inf;
Results(input.Itmax).DistrictMoves = [-inf -inf];
Results(input.Itmax).T = -inf;
Results(input.Itmax).CountySum = -inf;
Results(input.Itmax).PI = -inf;
Results(input.Itmax).compchange = -inf;
Results(input.Itmax).county = -inf;

% Results(input.input.Itmax).WCompScore = -inf;
% Results(input.Itmax).maxdiff = -inf;
% Results(input.Itmax).Area=zeros(MAXDIST,1);
% Results(input.Itmax).Perim=zeros(MAXDIST,1);
% Results(input.Itmax).DEind = -inf;
% Results(input.Itmax).power = -inf;
Results(input.Itmax).prob = -inf;

if run==1
Data5 = fopen('Data5.txt','r');
Neighbors5 = fopen('Neighbors5.txt','r');
BoundaryLengths5 = fopen('BoundaryLengthskm5.txt','r');
```

```
NeighborIndices=fopen('NeighborIndices.txt','r');

sizeData = [7 MAXPREC];
sizeNhb = [MAXNHB MAXPREC];
sizeBL = [MAXNHB MAXPREC];
sizeNI = [MAXNHB MAXPREC];

DataT =fscanf(Data5, '%lf', sizeData);
NeighborsT = fscanf(Neighbors5,'%f',sizeNhb);
BLT = fscanf(BoundaryLengths5,'%f', sizeBL);
NIT = fscanf(NeighborIndices,'%d',sizeNI);

Data = DataT';
Neighbors = NeighborsT';
BL = BLT';
NI = NIT';

clear DataT NeighborsT BLT NIT Data5 Neighbors5 BoundaryLengths5
clear NeighborIndices sizeData sizeNhb sizeBL sizeNI

SCPrec = struct([]);
SavedSCPrec = struct([]);
if isempty(SCPrec)
    for i=1:MAXPREC
        SCPrec(i).county = Data(i,1); %County, numbered by odds up to 91
        SCPrec(i).geoID = Data(i,2); %Unique ID for each precinct.
        SCPrec(i).dist = Data(i,3); %Primary district each precinct resides
            in
        SCPrec(i).perimeter = Data(i,4); %Perimeter in km
        SCPrec(i).area = Data(i,5); %Area in km^2
        SCPrec(i).pop = Data(i,6); %Population (estimated)
        SCPrec(i).moe = Data(i,7); %Margin of Error for population
            (estimated)
        for j=1:MAXNHB
            SCPrec(i).nhb(j) = Neighbors(i,j); %Neighbors for each precinct
            SCPrec(i).nhblength(j) = BL(i,j); %Length of boundary with each
                neighbor
        end
    end
end

for i=1:MAXPREC
    for j=1:MAXNHB
        SCPrec(i).nhbindex(j)=NI(i,j); %Index for each neighbor
    end
end
end
%A = zeros(MAXPREC,MAXNHB);

%Loads Neighbor Indices. We comment this section out because we have
%pre-loaded the neighbor indices.

% for i=1:MAXPREC
%     for j=1:MAXNHB
%         for k=1:MAXPREC
%               if SCPrec(i).nhb(j) == SCPrec(k).geoID
%                   break
%               elseif SCPrec(i).nhb(j) == 0
%                   SCPrec(i).nhbindex(j)=0;
%                   A(i,j) = 0;
%                   break
%               end
%
%           end
%           if SCPrec(i).nhb(j) == SCPrec(k).geoID
%               SCPrec(i).nhbindex(j)=k;
%               A(i,j) = k;
%           elseif SCPrec(i).nhb(j) == 0
%               SCPrec(i).nhbindex(j)=0;
%               A(i,j)=0;
%           end
%       end
% end

TotalPerim=0;
for i=1:MAXPREC
    TotalPerim= TotalPerim + SCPrec(i).perimeter;
end

clear Neighbors Data BL NI

%% Finds Boundary Precincts
SCPrec = FindBoundaryPrecU(SCPrec, MAXPREC, MAXNHB);

%% Establishes Maps
if run==1
if exist('CountyMap.mat','file')==2
    load('CountyMap.mat');
else
    countymap = shaperead('SC_Counties_Export.shp');
end
if exist('PrecMap.mat','file')==2
    load('PrecMap.mat');
else
    precmap = shaperead('Statewide_Precincts_2016.shp');
end

table = struct2table(precmap);
sortedT = sortrows(table, 'PrecinctID');
sortedS = table2struct(sortedT);
if input.comments ==1
    fprintf('It has been sorted!');
end

% This loop is used to assign unsorted districts to the struct 'sortedS'
% for i=1:MAXPREC+1
%     if str2double(sortedS(i).PrecinctID) == 99999
%         sortedS(i).Dist = 0;
%         %        fprintf('This happened at i= %f',i);
%         continue;
%       end
%       for j=1:MAXPREC+1
%           if str2double(sortedS(i).PrecinctID) == SCPrec(j).geoID
%               sortedS(i).Dist = SCPrec(j).dist;
%               break;
%           end
%       end
% end

%Quicker loop to assign districts to the struct 'sortedS'
for i=1:MAXPREC
    if str2double(sortedS(i).PrecinctID)~=SCPrec(i).geoID
        fprintf('Not equal at %d\n',i);
    else
        sortedS(i).Dist = SCPrec(i).dist;
    end
end
sortedS(MAXPREC+1).Dist = 0;
end

if input.video ==1
    precspec = makesymbolspec('Polygon',...
        {'Dist',0, 'FaceColor','k'},...
        {'Dist',1, 'FaceColor','r'},...
        {'Dist',2, 'FaceColor','g'},...
        {'Dist',3, 'FaceColor','c'},...
        {'Dist',4, 'FaceColor','y'},...
        {'Dist',5, 'FaceColor','m'},...
        {'Dist',6, 'FaceColor','b'},...
        {'Dist',7, 'FaceColor','w'});
    countyspec = makesymbolspec('Polygon', {'Default', 'FaceAlpha',0},...
        {'Default', 'LineWidth',2});
    figure
    MVcount =1;
end

if input.figures >=1
    precspec = makesymbolspec('Polygon',...
        {'Dist',0, 'FaceColor','k'},...
        {'Dist',1, 'FaceColor','r'},...
        {'Dist',2, 'FaceColor','g'},...
        {'Dist',3, 'FaceColor','c'},...
        {'Dist',4, 'FaceColor','y'},...
        {'Dist',5, 'FaceColor','m'},...
        {'Dist',6, 'FaceColor','b'},...
        {'Dist',7, 'FaceColor','w'});
    countyspec = makesymbolspec('Polygon', {'Default', 'FaceAlpha',0},...
        {'Default', 'LineWidth',2});

    fig = fig+1;
    figure(fig)
    mapshow(sortedS,'SymbolSpec',precspec);
    title('Current SC Precinct Map');
    hold on
    mapshow(countymap,'SymbolSpec',countyspec);
    drawnow
end

if input.BM ==1
    for i=1:MAXPREC
        sortedS(i).boundary=SCPrec(i).boundary;
    end
    sortedS(MAXPREC+1).boundary=2;


    boundaryspec = makesymbolspec('Polygon', {'boundary',0,
       'FaceColor','b'},...
        {'boundary',1, 'FaceColor','r'},...
        {'boundary',2, 'FaceColor','k'});
    fig = fig+1;
    figure(fig)
    mapshow(sortedS, 'SymbolSpec',boundaryspec);
    drawnow
end

if input.saving ==1
    String = [directoryname,'/BegMap_',currdate,'_run',runnum,'.mat'];
    save(String,'sortedS');
end

%% Calculates Polsby-Popper score for each district.
[DistArea, DistPerim, CurrentPP, DistPop] = PolsbyPopper(SCPrec,
   MAXDIST,MAXPREC,MAXNHB);
InitDP = DistPerim;
InitDA = DistArea;
CurrentinvPP = 1./CurrentPP; %We work with inverse Polsby-Popper
AvgCurrentinvPP = mean(CurrentinvPP);
MaxPP = AvgCurrentinvPP; %Initial maximum Polsby-Popper score achieved
TargetPop = mean(DistPop);

%% Calculates number of precincts in each county/district intersection
PinCperDist = zeros(MAXCOUNTY,MAXDIST);
for i=1:MAXPREC
    j=ceil(SCPrec(i).county/2);
    k=SCPrec(i).dist;
    PinCperDist(j,k) = PinCperDist(j,k)+1;
end

count=0;
it=1;
trueit=1;
vlc=1;
if input.CS ==2
    flag=0;
end

%% Main loop
while it<=input.Itmax
    if input.video ==1
        clf %clear figure
```

```
end
if input.CS ==0
    if it<input.Itmax/2
        T=input.initT;
    else
        T=.1;
    end
elseif input.CS ==1
    count=count+1;
    if T ==-inf
        T=input.initT;
    end
    if count>input.MCL
        T=input.alpha*T;
        count=1;
    end
elseif input.CS ==2
    if flag ==0 %We haven't reached peak yet
        T = input.initT;
    elseif flag ==1 %We've reached peak, and are now descending
        count=count+1;
        if count>input.MCL
            T=input.alpha*T;
            count=0;
        end
    else
        error('flag undefined');
    end
end

%prevents T from getting too small.
if T <0.05
    T =0.05;
end

%Increases strength of population control as algorithm progresses
if it/input.Itmax <0.2
    input.power = 1;
elseif it/input.Itmax <0.4
    input.power = 2;
elseif it/input.Itmax <0.6
    input.power = 4;
elseif it/input.Itmax <0.8
    input.power = 8;
elseif it/input.Itmax <0.9
    input.power = 16;
elseif it/input.Itmax <0.95
    input.power = 32;
elseif it/input.Itmax <0.975
    input.power = 64;
else
    input.power = 128;
end

if it-1 == ceil(input.Itmax/2) && input.saving ==1
```

```
        String = [directoryname,'/MidMap_',currdate,'_run',runnum,'.mat'];
        save(String,'sortedS');
    end

    if input.BM ==1 && it == floor(input.Itmax/2)
        for i=1:MAXPREC
            sortedS(i).boundary=SCPrec(i).boundary;
        end
        fig = fig+1;
        figure(fig)
        mapshow(sortedS,'SymbolSpec',boundaryspec);
        drawnow
    end

    % Saves a map at 20%, 40%, 60%, 80% and 90% of the way through
    if input.figures ==2 &&
        ismember((it-1)/input.Itmax,[0.2,0.4,0.6,0.8,0.9])
        fig=fig+1;
        figure(fig)
        mapshow(sortedS,'SymbolSpec',precspec);
        hold on
        mapshow(countymap,'SymbolSpec',countyspec);
        str = sprintf('SC Precinct Map after %d Iterations', it);
        title(str);
        drawnow
    end

    if input.comments ==1
        fprintf('\n');
    end

    %% Delta E Calculation Loop
    DEind = -inf;

    %The while loop is used only when the entire list of original boundary
        precincts breaks adjacency
    while DEind==-inf
        %Selects n random precincts on district boundaries
        [BoundaryPrecInd,NhbDist] = BPSelector(SCPrec,input.n,MAXPREC,0);

        HypDistPP = zeros(MAXDIST,input.n);
        HypDistArea = zeros(MAXDIST,input.n);
        HypDistPerim = zeros(MAXDIST,input.n);
        HypDistPop = zeros(MAXDIST,input.n);
        DeltaE = zeros(input.n,1);
        DeltaComp = zeros(input.n,1);
        PopScale = zeros(input.n,1);

        % Calculates Delta E for the n precincts
        for i=1:input.n
            [HypDistArea(:,i),HypDistPerim(:,i),HypDistPP(:,i),
                HypDistPop(:,i)] = ...
                 PPUpdate(SCPrec,BoundaryPrecInd(i),NhbDist(i), DistArea,
                     DistPerim, DistPop, MAXNHB);
            invDistPP = 1./HypDistPP;
            compNew=mean(invDistPP(:,i));
            cc= SCPrec(BoundaryPrecInd(i)).county;
            ccu=cc/2+0.5;
            [DeltaComp(i),DeltaE(i)] = DeltaEComp(PinCperDist(ccu,:), ...
                SCPrec(BoundaryPrecInd(i)).dist, NhbDist(i), compNew, ...
                AvgCurrentinvPP, input.pref, input.Exp, input.Exp2,
                    input.damping);
            PopScale(i) = PopComp(SCPrec(BoundaryPrecInd(i)).dist,
                NhbDist(i),...
                DistPop, HypDistPop(:,i),TargetPop,input.power,input.PPL);
        end

        if input.damping ==0
        maxDE = max(abs(DeltaE));
        DeltaE = DeltaE/maxDE;
        end

        %Picks a precinct to swap over district lines. If all precinct
        %choices break adjacency, we retry (DEind = -inf if all choices
        %break adjacency).
        [DEind, MovingPrecIndex, NewDist, prob] =
            Pickswap(BoundaryPrecInd,...
            NhbDist, SCPrec, DeltaE, PopScale, T, input.n, MAXNHB,
                input.comments );
    end

    %% Updates

    %Updates the district area, perimeter, and population after the moving
    %precinct has been chosen
    OldDistArea=DistArea;
    OldDistPerim=DistPerim;
    DistArea=HypDistArea(:,DEind);
    DistPerim=HypDistPerim(:,DEind);
    if sum(DistPerim) > TotalPerim
        error('Max perimeter exceeded');
    end
    DistPop = HypDistPop(:,DEind);
    %    Results(trueit).Area=DistArea;
    %    Results(trueit).Perim=DistPerim;
    Results(trueit).DistPop=DistPop;
    %    Results(trueit).DEind = DEind;
    Results(trueit).PopDiff = 100*(DistPop-TargetPop)/TargetPop;
    Results(trueit).PopScale = PopScale(DEind);
    Results(trueit).DeltaE = DeltaE(DEind);
    %    Results(trueit).power = power;
    Results(trueit).prob = prob;
    Results(trueit).DeltaComp = DeltaComp(DEind);
    %    Results(trueit).maxdiff = max(abs(Results(trueit).PopDiff));

    %Calculates current Polsby-Popper Score
    CurrentPP = 4*3.141592*OldDistArea./(OldDistPerim.^2);
    AvgCurrentinvPP = mean(1./CurrentPP);

    %Modifies the chosen prencinct's district
```

```
OldDist=SCPrec(MovingPrecIndex).dist;
SCPrec(MovingPrecIndex).dist=NewDist;
sortedS(MovingPrecIndex).Dist=NewDist;

%Modifies PinCperDist
cc = SCPrec(MovingPrecIndex).county;
ccu = cc/2+0.5;
PinCperDist(ccu,OldDist) = PinCperDist(ccu,OldDist)-1;
PinCperDist(ccu,NewDist) = PinCperDist(ccu,NewDist)+1;

%Modifies CountySum
CountySum = nnz(PinCperDist)-MAXCOUNTY;
Results(trueit).CountySum = CountySum;

%Modifies which precincts are on boundaries
SCPrec = BPUpdate(MovingPrecIndex, SCPrec, MAXNHB);

NewDistPP = 4*3.141592*DistArea./(DistPerim.^2);
AverageNewPP = mean(1./NewDistPP);
change = AverageNewPP - AvgCurrentinvPP;

if input.comments==1
    fprintf('We are on iteration %d. T = %f\n',it,T);
    fprintf('We moved precinct %d (index %d) from district %d to
        district %d.\n',...
         SCPrec(MovingPrecIndex).geoID, MovingPrecIndex, OldDist,
             NewDist);
    fprintf('The new compactness score is %f. This is a change of %f
        from the old score.\n',...
         AverageNewPP,change);
elseif input.comments ==0 && floor(trueit/500)== trueit/500
    fprintf('We are on iteration %d. T = %f\n',it,T);
end

Results(trueit).CompactnessScore=AverageNewPP;
Results(trueit).MovedPrec = SCPrec(MovingPrecIndex).geoID;
Results(trueit).PI = MovingPrecIndex;
Results(trueit).DistrictMoves(1) = OldDist;
Results(trueit).DistrictMoves(2) = NewDist;
Results(trueit).T = T;
%Results(trueit).count=count;
Results(trueit).compchange=change;
Results(trueit).county = SCPrec(MovingPrecIndex).county;


%      if it==1
%         Results(trueit).PerimChange = Results(trueit).Perim-InitDP;
%      else
%         Results(trueit).PerimChange = Results(trueit).Perim-
    Results(it-1).Perim;
%      end

if input.CS ==2 && flag ==0
    if trueit/500 == floor(trueit/500) && trueit>=1000
        PastScores = [Results(trueit-999:trueit).CompactnessScore];
        x=linspace(1,1000,1000);
        p1=polyfit(x,PastScores,1);
        %Swtiches to decreasing temperature if we've reached peak or we
        %are halfway through
        if (p1(1)<0.00005 && p1(1) >-0.00005) || it/input.Itmax >=.5
            flag=1;
            greenmarker =trueit;
            fprintf('We switched to decreasing temperature at iteration
                %d\n',it);
        end
    end
end

if it > .9*input.Itmax && max(abs(Results(trueit).PopDiff))<=input.PPL
    SavedSCPrec = SCPrec;
    FinalMapIt = trueit;
end

if AverageNewPP > MaxPP
    MaxPP = AverageNewPP;
end

%Advances the code to the next benchmark if we aren't improving
%compactness
if floor(it/1000) == it/1000 && AverageNewPP<0.5*MaxPP &&trueit>=2000
    && it/input.Itmax<0.9
    PastScores = [Results(trueit-1999:trueit).CompactnessScore];
    x=linspace(1,2000,2000);
    p2=polyfit(x,PastScores,1);
    fprintf('We are checking slope!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
        The slope is %f.\n',p2(1));
    if p2(1)<input.eps && p2(1) >-input.eps
        oldit=it;
        it = floor(input.Itmax*ceil(it/input.Itmax*10)/10);
        fprintf('Slope is small enough. Old it = %d. New it =
            %d.\n',oldit, it);
        if it ~= oldit
            count=0;
            vertlines(vlc) = trueit;
            itvals(vlc) = it/1000;
            vlc=vlc+1;
        end

        if input.CS ==1
            T = input.initT*input.alpha^(it/input.MCL);
        elseif input.CS ==2
            T = input.initT*input.alpha^((it-greenmarker)/input.MCL);
        end
    end
end

if input.video ==1
    if floor(it/input.videospace) == it/input.videospace || it ==1
        mapshow(sortedS,'SymbolSpec',precspec);
        hold on
```

```
            mapshow(countymap,'SymbolSpec',countyspec);
            str = sprintf('SC Precinct Map after %d Iterations', it);
            title(str);
            movieVector(MVcount) = getframe;
            MVcount=MVcount+1;
        end
    end

    it=it+1;
    trueit = trueit+1;
end

if input.optcomp ==1 %DOESN'T WORK
    [SCPrec,Results] = MaxComp(SCPrec,input,TargetPop);
end


Results(trueit:input.Itmax)=[]; % Empties the zeros left in the Results
   struct

if input.video ==1
    mywriter = VideoWriter(['Map_',currdate],'MPEG-4');
    mywriter.FrameRate = 2;
    mywriter.Quality = 100;
    open(mywriter);
    writeVideo(mywriter,movieVector);
    close(mywriter);
end

distprec= zeros(MAXPREC, MAXDIST);
row=ones(1,MAXDIST);

% Creates matrix detailing which precincts are in which districts
for i=1:MAXPREC
    for j=1:MAXDIST
        if SCPrec(i).dist ==j
            distprec(row(j),j) = SCPrec(i).geoID;
            row(j)=row(j)+1;
        end
    end
end
distprec(max(row):MAXPREC,:) = [];

%Presents the map that has correct population constraints.
if isempty(SavedSCPrec)
    fprintf('None of the final maps had the correct population requirements
        :(\n');
    FinalMapIt =trueit-1;
else
    for i=1:MAXPREC
        sortedS(i).Dist = SavedSCPrec(i).dist;
    end
    fprintf('This final map occurred at iteration %d\n',FinalMapIt);
end
```

```
NewDistPP
FinalCompScore = Results(FinalMapIt).CompactnessScore
AverageDeltaE = mean([Results.DeltaE])
AverageComp = mean([Results.DeltaComp])

if input.figures >=1
    % Displays final map
    fig = fig+1;
    figure(fig)
    mapshow(sortedS,'SymbolSpec',precspec);
    hold on
    mapshow(countymap,'SymbolSpec',countyspec);
    str = sprintf('SC Precinct Map after %d Iterations', FinalMapIt);
    title(str);
    drawnow

    % Displays compactness score over time
    fig = fig+1;
    figure(fig)
    if exist('itvals','var') ==1
        for i=1:length(itvals)
            list{1,i} = num2str(itvals(i));
        end
    end
    plot([Results.CompactnessScore]);
    if exist('vertlines','var')==1
        vline(vertlines,'r');
        input.vertlines = vertlines;
    end
    if exist('greenmarker','var')==1
        vline(greenmarker,'g');
        input.greenmarker = greenmarker;
    end
    title('Inverse Polsby-Popper Score by Iteration');
    xlabel('Iteration');
    ylabel('Inverse Polsby-Popper Score');

    % Displays CountySum values over time
    fig=fig+1;
    figure(fig)
    plot([Results.CountySum]);
    title('County Sum by Iteration');
    xlabel('Iteration');
    ylabel('County Sum');

    %Displays histogram of which counties had precincts swaps in them the
    %most often.
    fig=fig+1;
    figure(fig)
    histogram([Results.county]);
    title('Counties with precinct swaps');
    xlabel('County Number');
    ylabel('Number of swaps in this county');

    % Displays histogram of which precincts were swapped most often
    fig=fig+1;
    figure(fig)
    histogram([Results.PI],MAXPREC);
    title('Histogram of Precinct swaps');
    xlabel('Precinct ID');
    ylabel('Number of swaps');
end

if input.BM ==1
    for i=1:MAXPREC
        sortedS(i).boundary=SCPrec(i).boundary;
    end
    fig = fig + 1;
    figure(fig)
    mapshow(sortedS,'SymbolSpec',boundaryspec);
    drawnow
end

if input.saving ==1
    String = [directoryname,'/Results_',currdate,'_run',runnum,'.mat'];
    save(String,'Results');

    String = [directoryname,'/FinMap_',currdate,'_run',runnum,'.mat'];
    save(String,'sortedS');
end
clear str oldit ans cc ccu i j k it NewDist OldDist NhbDist vlc x
clear HypDistArea HypDistPerim HypDistPop HypDistPP MovingPrecIndex
clear table SavedScores p2 OldDistArea OldDistPerim count prob change
clear compNew DEind
end
```

The function FindBoundaryPrecU examines all of South Carolina's precincts and determines which of them are on district boundaries. It saves the result in SCPrec, which is the struct containing the data about the precincts.

Listing 2: FindBoundaryPrecU function

```
function [SCPrec] = FindBoundaryPrecU(SCPrec, MAXPREC, MAXNHB )
% This function examines all precincts and determines which ones are on
    district boundaries

for i=1:MAXPREC
    SCPrec(i).boundary=0; % Indicates that the precinct is not on a
        district boundary
end

for i=1:MAXPREC
    if SCPrec(i).boundary==1
        continue
    end

    Dist = SCPrec(i).dist;
    for k=1:MAXNHB
        if SCPrec(i).nhbindex(k)==0
           break
        end
        NI = SCPrec(i).nhbindex(k);
        NDist = SCPrec(NI).dist;

        % If the precinct is on a district boundary, its boundary field is
        % changed
        if Dist ~= NDist
            SCPrec(i).boundary=1;
            SCPrec(NI).boundary=1;
        end
    end

end

end
```

The PolsbyPopper function will calculate the Polsby-Popper score for each district as well as each district's area, perimeter, and population. To calculate perimeter, we added together the perimeter of each precinct in a district and then subtracted the perimeter that was on the interior of the district.

Listing 3: PolsbyPopper function

```
function [DistArea, DistPerim, DistPP, DistPop] = PolsbyPopper(SCPrec,
    MAXDIST,MAXPREC,MAXNHB)
%PolsbyPopper calculates the Polsby-Popper score for each district

AreaSum = zeros(MAXDIST,1);
TotalPerim = zeros(MAXDIST,1);
IntPerim = zeros(MAXDIST,1);
DistPop = zeros(MAXDIST,1);

for i=1:MAXDIST
    for j=1:MAXPREC
        if SCPrec(j).dist==i
            DistPop(i) = DistPop(i) + SCPrec(j).pop;
            AreaSum(i)=AreaSum(i)+SCPrec(j).area;
            TotalPerim(i)=TotalPerim(i)+SCPrec(j).perimeter;
            for k=1:MAXNHB
                CNI = SCPrec(j).nhbindex(k);
                if CNI ==0
                    break
                elseif SCPrec(j).dist == SCPrec(CNI).dist
                    IntPerim(i)=IntPerim(i)+SCPrec(j).nhblength(k);
                end
            end
        end
    end
end

OuterPerim=TotalPerim-IntPerim;

DistPP = 4*3.141592*AreaSum./(OuterPerim.^2);
DistArea=AreaSum;
DistPerim=OuterPerim;
end
```

BPSelector (Boundary Precinct Selector) selects n precincts on district boundaries. It will also find the district that neighbors each of those precincts. If there are two or more districts that neighbor a single precinct, we will pick one at random (the random number is given on line 19). Everything after line 28 was not used for this project.

Listing 4: BPSelector function

```
function [ BoundaryPrecInd, NhbDist ] = BPSelector( SCPrec, n, MAXPREC,
    newcode)
%This selects n precincts on district boundaries

% Case where we use GC_4_0
if newcode ~= 5

    boundaryprecs = find([SCPrec.boundary]);

    BoundaryPrecInd = datasample(boundaryprecs,n);

    NhbDist = zeros(n,1);
    BPind = zeros(n,1);
    for i=1:n
        flag=0;
        BPind(i) = BoundaryPrecInd(i);
        BPDist = SCPrec(BPind(i)).dist;
        nhbcount = nnz([SCPrec(BPind(i)).nhbindex]);
        while flag ==0
            r=randi(nhbcount);
            NI = SCPrec(BPind(i)).nhbindex(r);
            if BPDist ~= SCPrec(NI).dist
                flag=1;
                NhbDist(i)=SCPrec(NI).dist;
            end
        end
    end

% Case where we use GC_5_0
else
    flag=0;
    while flag == 0
        HypPrecInd = randi(MAXPREC);
        if SCPrec(HypPrecInd).boundary ==1
            flag =1;
            BoundaryPrecInd =HypPrecInd;
        end
    end

    flag=0;
    BPind = BoundaryPrecInd;
    BPDist = SCPrec(BPind).dist;
    nhbcount = nnz([SCPrec(BPind).nhbindex]);
    while flag ==0
        r=randi(nhbcount);
        NI = SCPrec(BPind).nhbindex(r);
        if BPDist ~= SCPrec(NI).dist
            flag=1;
            NhbDist=SCPrec(NI).dist;
        end
    end

end


end
```

PPUpdate (Polsby-Popper Update) is a function that calculates hypothetical district areas, perimeters, Polsby-Popper scores, and populations for a given precinct swap. By inputting the candidate precinct (MovingPrec) and the district we are proposing it move to (NewDist), we can easily update each of the items above.

Listing 5: PPUpdate function

```
function [ DistArea, DistPerim, DistPP, DistPop ] = PPUpdate(
    SCPrec,MovingPrec,NewDist, DistArea, DistPerim, DistPop, MAXNHB )
% Updates the Polsby-Popper scores for each district without recomputing
% everything

OldDist = SCPrec(MovingPrec).dist;

%Adjust population of old and new district
DistPop(NewDist) = DistPop(NewDist)+SCPrec(MovingPrec).pop;
DistPop(OldDist) = DistPop(OldDist)-SCPrec(MovingPrec).pop;

%Adjusts area of the new district
DistArea(NewDist)=DistArea(NewDist)+SCPrec(MovingPrec).area;
DistArea(OldDist)=DistArea(OldDist)-SCPrec(MovingPrec).area;

% Preliminarily adjusts perimeter of the new district
DistPerim(NewDist)=DistPerim(NewDist)+SCPrec(MovingPrec).perimeter;
DistPerim(OldDist)=DistPerim(OldDist)-SCPrec(MovingPrec).perimeter;
if DistPerim(OldDist) <0
    error('Perimeter is less than 0 (OldDist)\n');
elseif DistPerim(NewDist) <0
    error('Perimeter is less than 0 (NewDist)\n');
end

% Runs through neighbors of MovingPrec to adjust the perimeter
for k=1:MAXNHB
    if SCPrec(MovingPrec).nhbindex(k)==0
        break;
    else
        NI = SCPrec(MovingPrec).nhbindex(k);
        %fprintf('MovingPrec = %d, NI = %d,
            SCPrec(MovingPrec).nhblength(k)=%f\n',MovingPrec,NI,SCPrec(MovingPrec).nhblength(
        if SCPrec(NI).dist ==OldDist
            DistPerim(OldDist)=DistPerim(OldDist)+2*SCPrec(MovingPrec).nhblength(k);
            if DistPerim(OldDist) <0
                error('Perimeter is less than 0 (OldDist)\n');
            end
        elseif SCPrec(NI).dist == NewDist
            DistPerim(NewDist)=DistPerim(NewDist)-2*SCPrec(MovingPrec).nhblength(k);
            if DistPerim(NewDist) <0
                error('Perimeter is less than 0 (NewDist)\n');
            end
            %This multiplication by 2 occurs because we have doubled up on
            %the perimeter between the MovingPrec and NewDist
        end
    end
end

DistPP= 4*3.141592*DistArea./(DistPerim).^2;


end
```

DeltaEComp (ΔE Computation) is a function which computes ΔE values for the proposed precinct swap. This formulation is described in detail in subsection 4.2.3. Everything past line 29 was not used in this project.

Listing 6: DeltaEComp function

```
function [DeltaComp, DeltaE ] = DeltaEComp( PinCrow, OldDist, NewDist,
    compN, compC, pref, Exp, Exp2, damp)
%DeltaEComp Computes the DeltaE score for the proposed swap
PCOld = PinCrow(OldDist);
PCNew = PinCrow(NewDist);
DeltaComp = compN-compC;

% if DeltaComp <-1
%    fprintf('DeltaComp was too negative. DeltaComp = %f\n', DeltaComp);
%    DeltaComp = -1;
% elseif DeltaComp >1
%     fprintf('DeltaComp was too positive. DeltaComp = %f\n', DeltaComp);
%     DeltaComp = 1;
% end
x=PCOld-PCNew;

if pref == 0 %Case where swaps that improve compactness are incentivized

    if sign(x)==sign(DeltaComp)
        S = 1+(x^2)^(1/3);
    else
        S = 1/(1+(x^2)^(1/3));
    end

    if DeltaComp<0
        DeltaE=S^Exp2*(-DeltaComp+1)^Exp*sign(DeltaComp);
    elseif DeltaComp >=0
        DeltaE=S^Exp2*(DeltaComp+1)^Exp*sign(DeltaComp);
    end
elseif pref == 1 %Case where we incentivize swaps that make political
    boundaries nicer
    S = sign(x)*nthroot(x^2,3);
    if sign(x) == sign(DeltaComp)
        f = abs(DeltaComp)+1;
    else
        f = 1/(damp+abs(DeltaComp));
    end
    DeltaE=S*f^Exp; %As of 5-20-19, THIS WILL NOT WORK WITH EXP =/= 1.
end

end
```

PopComp (Population Computation) is a function which calculates the population scaling factor S_j mentioned in Equation (4.1). Let's suppose that the proposed precinct swap is from "old district" to "new district." In this function, DistPop is a vector containing the population of each district *after* the proposed swap. OldDistPop is the vector containing the population of each district *before* the proposed swap. Then, we calculate the population scaling variable as follows:

- We start by calculating a variable q for the old district. The variable q is the number of percentage points that the population of the old district is away from the ideal population window.
  - If the old district's population is *above* the ideal population window, then q will be positive. This is good, since this proposed swap will reduce the population of the old district.
  - If the old district's population is *below* the ideal population window, then q will be negative. This is bad, because the proposed swap will reduce the population of the old district.
  - If the old district's population is within the ideal population window, then q will be 0.
- We then calculate a second q for the new district, in a mirrored way.
  - If the new district's population is *above* the ideal population window, then q will be negative.
  - If the new district's population is *below* the ideal population window, then q will be positive.
  - If the new district's population is within the ideal population window, then q will be 0.
- We add the two q values together.
- We now calculate a variable y based on the sign of q.
  - If $q > 0$, then $y = 0.1 * q$.
  - If $q == 0$, then $y = 0$.
  - If $q < 0$, then $y = 0.9^{-q} - 1$.
- Finally, we calculate the population scaling variable. $S_j = (1 + y)$ and $S_j^p = (1 + y)^p$.

The idea behind the calculation of y is that if q is positive, then $1 + y \in (1, \infty)$, so the scaling variable will be greater than one. Conversely, if $q < 0$, then $1 + y \in (0, 1)$, so the scaling variable will also be in $(0, 1)$.

Listing 7: PopComp function

```
function [PopScale] = PopComp(OldDist, NewDist,OldDistPop, DistPop,
    TargetPop, power,PPL)
% PopComp computes a number which is a multiplier for the probability
% function. If the swap gets the districts closer to the target
% populations, the scaling is an increase, otherwise, the probability of
% making that swap diminishes.

uppop = (1+0.01*PPL)*TargetPop;
lwrpop = (1-0.01*PPL)*TargetPop;

%Establishes q for OldDist
if DistPop(OldDist) >uppop
    q=(DistPop(OldDist)-uppop)/(0.01*TargetPop); %positive q
elseif DistPop(OldDist) <= uppop && DistPop(OldDist) >= lwrpop &&
    OldDistPop(OldDist) <= uppop && OldDistPop(OldDist) >=lwrpop
    q=0;
elseif DistPop(OldDist) <= uppop && DistPop(OldDist) && OldDistPop(OldDist)
    > uppop
    q=(OldDistPop(OldDist)-uppop)/(0.01*TargetPop); %positive q
elseif DistPop(OldDist) <= uppop && DistPop(OldDist) && OldDistPop(OldDist)
    < lwrpop
    q=(OldDistPop(OldDist)-lwrpop)/(0.01*TargetPop); %negative q
elseif DistPop(OldDist) <lwrpop
    q=(DistPop(OldDist)-lwrpop)/(0.01*TargetPop); %negative q
else
    error('q was not assigned (OldDist)');
end

%Modifies q for NewDist
if DistPop(NewDist) >uppop
    q=q-(DistPop(NewDist)-uppop)/(0.01*TargetPop); %subtracting positive
        number
elseif DistPop(NewDist) <= uppop && DistPop(NewDist) >= lwrpop &&
    OldDistPop(NewDist) <= uppop && OldDistPop(NewDist) >=lwrpop
    q=q-0;
elseif DistPop(NewDist) <= uppop && DistPop(NewDist) >= lwrpop &&
    OldDistPop(NewDist) > uppop
    q=q-(OldDistPop(NewDist)-uppop)/(0.01*TargetPop); %subtracting positive
        number
elseif DistPop(NewDist) <= uppop && DistPop(NewDist) >= lwrpop &&
    OldDistPop(NewDist) < lwrpop
    q=q-(OldDistPop(NewDist)-lwrpop)/(0.01*TargetPop); %subtracting
        negative number
elseif DistPop(NewDist) <lwrpop
    q=q-(DistPop(NewDist)-lwrpop)/(0.01*TargetPop); %subtracting negative
        number
else
```

```
    error('q was not assigned (NewDist)');
end

if q>0
    y=0.1*q;
elseif q==0
    y=0;
elseif q<0
    y=(0.9)^(-q)-1;
end

PopScale = (1+y)^power;

%fprintf('q is %f and y is %f, resulting in PopScale = %f\n',q,y,PopScale);
```

PickSwap is the function that assigns weights to each of the n boundary precincts, and then picks one to move over district lines. The equation dictating how weights are assigned can be found in equation (4.1). In order to save computation time, we only check the contiguity of the districts *after* we have picked a precinct to swap over district lines. If that precinct would break contiguity, we choose another one. The call to the function which checks contiguity is found on line 54.

Listing 8: Pickswap function

```
function [ DEind, MovingPrecInd,NewDist, prob ] = Pickswap(BoundaryPrecInd,
    NhbDist, SCPrec, DeltaE, PopScale, T, n, MAXNHB, comments )
%Picks a prencinct to swap over district lines

MovingPrecInd = -inf;
NewDist = -inf;

% for i=1:n
%     if PopScale(i)<10^-50
%         PopScale(i)=0;
%     end
% end
Probability = PopScale./(1+exp(DeltaE/T));

%Forces exceptionally small probabilities to be zero
for i=1:n
    if Probability(i) <0
        error('A probability is less than 0');
    elseif Probability(i) == inf
        fprintf('A probability was infinity\n');
        MovingPrecInd = BoundaryPrecInd(i);
        DEind=i;
        NewDist = NhbDist(i);
        prob=100;
        return
    elseif Probability(i)<=10^(-50)
%         Probability(i)=0;
    end
end




ProbabilityInts = zeros(n,1);
ProbabilityInts(1) = Probability(1);
for i=2:n
    ProbabilityInts(i) = Probability(i)+ProbabilityInts(i-1);
end



Adj = 0; %Starting value.
while Adj ==0
    if ProbabilityInts(n) <= 0
```

```
        fprintf('All swap choices break adjacency; choosing new boundary
            precincts.\n');
        DEind=-inf;
        prob=-inf;
        return;
    end
    RandomNumber = rand*ProbabilityInts(n);
    maxprob=100*max(Probability/sum(Probability));

    if RandomNumber <= ProbabilityInts(1) && RandomNumber >0
        MovingPrecInd=BoundaryPrecInd(1);
        Adj = CheckAdjacency(MovingPrecInd, SCPrec, MAXNHB);
        if Adj == 0
            if comments ==1
                fprintf('FIRST LOOP: Our original precinct choice (geoID
                    %d, index %d, dist %d) violated adjacency of the old
                    district\n',SCPrec(MovingPrecInd).geoID, MovingPrecInd,
                    SCPrec(MovingPrecInd).dist);
                fprintf('RandomNumber is %.3f. In interval (0, %.3f].
                    Chance of choosing: %f percent.
                    \n',RandomNumber,ProbabilityInts(1),
                    100*ProbabilityInts(1)/ProbabilityInts(n));
            end
            Probability(1)=0;
            ProbabilityInts(1)=0; %Makes probability of choosing this swap 0
            for j=2:n
                ProbabilityInts(j)=ProbabilityInts(j)-ProbabilityInts(1);
                    %Adjusts all subsequent intervals
            end
            continue;
        end
        NewDist = NhbDist(1);
        if comments ==1
            fprintf('The probability of this pick is %f percent; largest
                was %f. \n',100*(ProbabilityInts(1))/ProbabilityInts(n),
                maxprob);
            fprintf('PopScale was %f\n',PopScale(1));
            fprintf('We are using the 1st element in BoundaryPrecInd\n');
        end
        i=1;
        break
    else
        for i=2:n
            if RandomNumber <= ProbabilityInts(i) && RandomNumber
                >ProbabilityInts(i-1)
                MovingPrecInd=BoundaryPrecInd(i);
                Adj = CheckAdjacency(MovingPrecInd, SCPrec, MAXNHB);
                if Adj == 0 %Considers what happens when adjacency is
                    broken from this swap
                    difference=ProbabilityInts(i)-ProbabilityInts(i-1);
                    if comments ==1
                        fprintf('Our original precinct choice (geoID %d,
                            index %d, dist %d) violated adjacency of the
                            old district\n',SCPrec(MovingPrecInd).geoID,
                            MovingPrecInd, SCPrec(MovingPrecInd).dist);
                        fprintf('RandomNumber is %.3f. In interval (%.3f,
                            %.3f]. Chance of choosing: %f
                            percent.\n',RandomNumber,ProbabilityInts(i-1),
                            ProbabilityInts(i),100*difference/ProbabilityInts(n));
                    end

                    ProbabilityInts(i)=ProbabilityInts(i-1); %Makes
                        probability of choosing this swap 0
                    Probability(i)=0;
                    for j=i+1:n
                        ProbabilityInts(j)=ProbabilityInts(j)-difference;
                            %Adjusts all subsequent intervals
                    end
                    break;
                end
                NewDist = NhbDist(i);
                if comments ==1
                fprintf('The probability of this pick is %f percent;
                    largest was %f.
                    \n',100*(ProbabilityInts(i)-ProbabilityInts(i-1))/ProbabilityInts(n),
                    maxprob);
                fprintf('PopScale was %f\n',PopScale(i));
                fprintf('We are using the %dth element in
                    BoundaryPrecInd\n',i);
                end
                break;
            end
        end
    end
end

DEind = i;
if i>1
    prob = 100*(ProbabilityInts(i)-ProbabilityInts(i-1))/ProbabilityInts(n);
else
    prob = 100*ProbabilityInts(1)/ProbabilityInts(n);
end

if MovingPrecInd == -Inf || NewDist == -Inf
    fprintf('ERROR. No Precinct has been chosen, likely because all weights
        are zero. Choosing a precinct at random: \n');
    DEind = randi(n);
end
end
```

The CheckAdjacency function checks whether districts are still contiguous after the swap is made. To do this, it is only necessary to analyze the district that the precinct originated from. Further, we need only ensure that the neighbors of the moved precinct remain connected after the swap. To check if the neighbors are still connected, we consider all of the neighbors' neighbors within the district excluding the moving precinct. If the set of all neighbors and neighbors' neighbors excluding the moving precinct are connected, then we say that the district would be contiguous after the swap. This is not perfect. There exist contrived examples where a precinct swap should be allowable, but is considered to break contiguity by this function. These examples seem quite rare and did not dramatically affect this analysis.

Listing 9: CheckAdjacency function

```
function [ Adj ] = CheckAdjacency( p, SCPrec, MAXNHB )
%Determines if all districts maintain contiguity after a precinct changes
%districts

        queue = zeros(50,1);
    checked = zeros(50,1);
        visited = zeros(50,1);
        adjMatrix = zeros(50,MAXNHB+1);
        i=1;
    qc=2;
    m=1;
    jmax=0;
        iterations=0;
        PNcount=0;
        count=0;
        D = SCPrec(p).dist;


        queue(1)=p;
        % This while loop will build a precinct which represents the
            subgraph
    % containing neighbors up to two edges away and in the same district
    while queue(1)~=0

                v=queue(1);
                l=1;
        while queue(l)~=0
                        queue(l)=queue(l+1); % Moves all elements in the
                            queue up one spot
                        l=l+1;
        end
                qc=qc-1;
                adjMatrix(i,1)=v;
                k=1;
                j=2;
        while SCPrec(v).nhbindex(k)~=0
%             fprintf('Index: %d. geoID: %d. Neighbor: %d. Neighbor Index:
    %d\n',v,SCPrec(v).geoID,k,SCPrec(v).nhbindex(k));
                        NI = SCPrec(v).nhbindex(k);
                        overlap=0;
                        n=1;
            while checked(n)~=0
                if NI==checked(n)
                                        overlap=1; % Checks to see if
                                            precinct has been counted as a
                                            neighbor already
                end
                                n=n+1;
            end

            if SCPrec(NI).dist==D && NI~=p
                                adjMatrix(i,j)=NI;
                if(iterations<PNcount+1 &&overlap==0)
                                        queue(qc)=NI; % Add new precinct
                                            into queue if it hasn't already
                                            been considered and isn't too
                                            far away.
                                        qc=qc+1;
                                        checked(m)=NI;
                                        m=m+1;
                    if(iterations==0)
                        PNcount=PNcount+1;
                    end
                end
                                j=j+1;
                if j>jmax
                    jmax=j;
                end
                if jmax >MAXNHB+1
                    %fprintf('You are trying to make jmax too large! v= %d
                        and NI = %d.\n',v,NI);
                    jmax=MAXNHB+1; %Something is wrong here.
                end


            end
                        k=k+1;
            if k>MAXNHB
                break;
            end
        end
                iterations=iterations+1;
                i=i+1;

    end

        imax=i;
        qc=2;
        k=2;
        queue(1)=adjMatrix(2,1);
        visited(1)=adjMatrix(2,1);
        % This loop does a breadth first search to determine if the graph
           is still connected when we remove the precinct p.
        while queue(1)~=0
                v=queue(1);
                l=1;
        while queue(l)~=0
                        queue(l)=queue(l+1);
                        l=l+1;
        end
                qc=qc-1;

        for b=1:imax
            if adjMatrix(b,1)==v
                                i=b;
            end
        end


        for j=1:jmax

%                       sprintf('Test 1: We are examining (%d,%d)\n',i,j);

            if adjMatrix(i,j)==0
               break;
            end
                        n=1;
                        overlap=0;
            while checked(n)~=0
                if adjMatrix(i,j)==checked(n)
                                        overlap = 1;
                                        break;
                end
                                n=n+1;
            end
                        m=1;
                        overlap2=0;
            while visited(m)~=0
                if adjMatrix(i,j)==visited(m)

                                        overlap2=1;
                                        break;
                end
                                m=m+1;
            end

            if overlap==1 && overlap2 ==0
                                queue(qc)=adjMatrix(i,j);
                                qc=qc+1;
                                visited(k)=adjMatrix(i,j);
                                k=k+1;
            end

        end
```

```
                count=count+1;
        if count>50
                        break;
        end
        end

    adjMatrix(1:imax,1:jmax);

    a=nnz(checked)+1;

        if k==a
        Adj=1;
        else
        Adj=0;
        end
end
```

BPUpdate (Boundary Precinct Update) is a function that updates the precincts that are on district boundaries after a swap has occurred. To update these, it is only necessary to update the precincts immediately neighboring the moved precinct.

Listing 10: BPUpdate function

```
function [SCPrec] = BPUpdate(MovedPrec, SCPrec, MAXNHB )
% This function updates all precincts on district boundaries

for k=1:MAXNHB
   if SCPrec(MovedPrec).nhbindex(k)==0
       break
   end
   flag=0;
   NI = SCPrec(MovedPrec).nhbindex(k);
   NDist = SCPrec(NI).dist;

   %Checking neighbors' neighbors
   for l=1:MAXNHB
      if SCPrec(NI).nhbindex(l)==0
          break
      end
      NNI = SCPrec(NI).nhbindex(l);
      NNDist = SCPrec(NNI).dist;
      if NDist ~= NNDist
          SCPrec(NI).boundary=1;
          flag=1;
          break
      end
   end

   % This removes the precinct with index NI from boundary list if it is no
   % longer on a district boundary
   if flag ==0
       SCPrec(NI).boundary=0;
   end
end

end
```

Appendix C MOSA2 Codes

```
    """
Created on Sun Jan 23 18:55:59 2022

@author: Blake Splitter with assistance from Amy Burton and Dr. Matthew
    Saltzman
"""

from copy import deepcopy
from multiprocessing.sharedctypes import Value
import sys
import datetime
import random
from inspect import signature
import networkx as nx
import numpy as np
import math
import pandas as pd
import statistics
import concurrent.futures
from gerrychain import Graph
from gerrychain.updaters import Tally, cut_edges
import warnings
import json
import matplotlib.pyplot as plt
import ast
import pickle
from pymoo.indicators.hv import HV
import secrets
import matplotlib.colors as mcolors

class input_vals:

    def __init__(self, fjs, in_p_f, in_n_f, in_c_f, in_vb_f, in_vr_f, dc,
                   nr, t, fin_t, tol, fin_tol, mfr, mff, nf, obj_s,
                       obj_s_f, nipt,
                    nm, mtu, par, psm, palp, alp, mpm, pw, nt, mm):
        lst = [fjs, in_p_f, in_n_f, in_c_f, in_vb_f, in_vr_f, dc,
                   nr, t, fin_t, tol, fin_tol, mfr, mff, nf, obj_s,
                       obj_s_f, nipt,
                    nm, mtu, par, psm, palp, alp, mpm, pw, nt, mm]
        if all(v is None for v in lst):
            self.default_user_input()
        else:
            self.file_json = fjs
            self.in_pop_field = in_p_f
            self.in_name_field = in_n_f
            self.in_county_field = in_c_f
            self.in_voteblue_field = in_vb_f
            self.in_votered_field = in_vr_f
            self.distcount = int(dc)
            self.num_recoms = int(nr)
            self.temp = float(t)
            self.final_temp = float(fin_t)
            self.tol = float(tol)
            self.final_tol = float(fin_tol)
            self.max_failed_recoms = int(mfr)
            self.max_failed_flips = int(mff)
            self.num_flips = int(nf)
            self.objective_scales = json.loads(obj_s)
            self.objective_scales_flips = json.loads(obj_s_f)
            self.num_its_per_temp = int(nipt)
            self.num_maps = int(nm)
            self.metrics_to_use = json.loads(mtu)
            self.parallel = eval(par.lower().capitalize())
            self.preselected_start_map = eval(psm.lower().capitalize())
            self.preselect_alphas = eval(palp.lower().capitalize())
            self.alphas = ast.literal_eval(alp)
            self.max_pop_multiplier = float(mpm)
            self.pop_window = float(pw)
            self.num_threads = int(nt)
            self.met_maxes = json.loads(mm)
            self.input_validation()

    def default_user_input(self):
        self.file_json = "./SC_Precincts_2_FeaturesToJSO.geojson"
        self.in_pop_field = "POPULATION"
        self.in_name_field = "OBJECTID"
        self.in_county_field = "COUNTY"
        self.in_voteblue_field = "PresBlue"
        self.in_votered_field = "PresRed"
        self.distcount = 7
        self.num_recoms = 6
        self.temp = 10
        self.final_temp = 0.005
        self.tol = 30
        self.final_tol = 0.1
        self.max_failed_recoms = 2
        self.max_failed_flips = 1
        self.num_flips = 15
        self.objective_scales = [20000, 0.5, 0.066641, 1, 25, 0.01]  #These
            are the scaling factors for pop_dev, compactness, efficiency
            gap, CDI, eGU, and MM respectively
        #self.objective_scales_flips = [3000, 0.01, 0.001, 0.1, 1, 0.001]
        #------------------------------pop, comp,  eg,   cdi, egu, mm
        self.objective_scales_flips = [3000, 0.05, 0.001, 0.05, 1, 0.001]
        self.num_its_per_temp = 1
        self.num_maps = 500
        self.metrics_to_use = [1, 1, 0, 0, 0, 0]  #Puts 1s in the spots for
            metrics we do want to use. [0]: Pop_dev, [1]: comp, [2]: eg,
            [3]: CDI, [4]: eGU, [5]: MM
        self.parallel = True
        self.preselected_start_map = False
        self.preselect_alphas = True
        self.alphas = [[0.5, 0.5, 0, 0, 0, 0], [0.25, 0.75, 0, 0, 0, 0]]
        #self.alphas = []
        self.max_pop_multiplier = 100
        self.pop_window = 0.01
        self.num_threads = 4
        self.met_maxes = [-1, 10, 0.24, 50, 500, 0.05] #Popdev max
            established later
        self.input_validation()

    def pop_max(self, stateG):
        """Finds met_maxes[0] (the population deviation maximum)"""
        if self.met_maxes[0] == -1:
            self.met_maxes[0] =
                sum(dict(stateG.nodes("POPULATION")).values())*0.4

    @property
    def coolingrate(self):
        """Calculates the necessary cooling rate to get from initial
            temperature to
        final temperature in the requested number of recom steps"""
        return (self.final_temp / self.temp) ** (1 / self.num_recoms)

    @property
    def total_obj_vals_entries(self):
        """Returns the number of objective value entries needed"""
        return (self.num_its_per_temp * (self.num_recoms *
            (self.num_flips+1))+1)


    @property
    def tol_coolingrate(self):
        """Calculates the cooling rate to get from starting tol to final
            tol"""
        return (self.final_tol / self.tol) ** (1 / self.num_recoms)

    @property
    def total_iterations(self):
        """Calculates the total number of loops in the SA algorithm we will
            do"""
        return self.num_recoms * self.num_its_per_temp

    def pop_multiplier(self, count):
        """Calculates how much we should multiply the population value by
            in this algorithm"""
        inv_pm = (self.total_iterations -
            count.recomcount)/self.total_iterations  #The fraction of
            iterations that have yet to be done
        try: pm = min(1/inv_pm, self.max_pop_multiplier)  #population
            multiplier is capped
        except ZeroDivisionError: pm = self.max_pop_multiplier
        if pm < 1: raise ValueError("The population multiplier is <1, which
            should not happen.")
        sq_pm = pm**2  #Squares the population multiplier to make it more
            important
        return sq_pm

    def input_validation(self):
        """Verifies that the input given is appropriate"""
        if self.alphas == None: raise ValueError("alphas should either be
            an empty list [] or a list of weight vectors")
        if not isinstance(self.file_json, str): raise ValueError("file_json
            is the wrong variable type")
        if not isinstance(self.in_pop_field, str): raise
            ValueError("in_pop_field is the wrong variable type")
        if not isinstance(self.in_name_field, str): raise
            ValueError("in_name_field is the wrong variable type")
        if not isinstance(self.in_county_field, str): raise
            ValueError("in_county_field is the wrong variable type")
        if not isinstance(self.in_voteblue_field, str): raise
            ValueError("in_voteblue_field is the wrong variable type")
        if not isinstance(self.in_votered_field, str): raise
            ValueError("in_votered_field is the wrong variable type")
        if not isinstance(self.distcount, (int, float)): raise
            ValueError("distcount is the wrong variable type")
        if not isinstance(self.num_recoms, (int, float)): raise
            ValueError("num_recoms is the wrong variable type")
        if not isinstance(self.temp, (int, float)): raise ValueError("temp
            is the wrong variable type")
        if not isinstance(self.final_temp, (int, float)): raise
            ValueError("final_temp is the wrong variable type")
        if not isinstance(self.tol, (int, float)): raise ValueError("tol is
            the wrong variable type")
        if not isinstance(self.final_tol, (int, float)): raise
            ValueError("final_tol is the wrong variable type")
        if not isinstance(self.max_failed_recoms, (int, float)): raise
            ValueError("max_failed_recoms is the wrong variable type")
        if not isinstance(self.max_failed_flips, (int, float)): raise
            ValueError("max_failed_flips is the wrong variable type")
        if not isinstance(self.num_flips, (int, float)): raise
            ValueError("num_flips is the wrong variable type")
        if not isinstance(self.objective_scales, list): raise
            ValueError("objective_scales is the wrong variable type")
        if not isinstance(self.objective_scales_flips, list): raise
            ValueError("objective_scales_flips is the wrong variable type")
        if not isinstance(self.num_its_per_temp, (int, float)): raise
            ValueError("num_its_per_temp is the wrong variable type")
        if not isinstance(self.num_maps, (int, float)): raise
            ValueError("num_maps is the wrong variable type")
        if not isinstance(self.metrics_to_use, list): raise
            ValueError("metrics_to_use is the wrong variable type")
        if not isinstance(self.parallel, bool): raise ValueError("parallel
            is the wrong variable type")
        if not isinstance(self.preselected_start_map, bool): raise
            ValueError("preselected_start_map is the wrong variable type")
        if not isinstance(self.preselect_alphas, bool): raise
            ValueError(f"preselect_alphas is type
            {type(self.preselect_alphas)}")
        if not isinstance(self.alphas, list): raise ValueError("alphas is
            the wrong variable type")
        if not isinstance(self.max_pop_multiplier, (int, float)): raise
            ValueError("max_pop_multiplier is the wrong variable type")
        if not isinstance(self.pop_window, (int, float)): raise
            ValueError("pop_window is the wrong variable type")
        if not isinstance(self.num_threads, (int, float)): raise
            ValueError("num_threads is the wrong variable type")

        for variable in [self.distcount, self.num_recoms,
            self.max_failed_recoms, self.max_failed_flips, self.num_flips,
            self.num_its_per_temp, self.num_maps, self.num_threads]:
             self.check_if_int(variable)

        for variable in [self.distcount, self.num_recoms, self.temp,
            self.final_temp, self.tol, self.final_tol,
            self.max_failed_recoms, self.max_failed_flips, self.num_flips, \
                          self.num_its_per_temp, self.num_maps,
                              self.max_pop_multiplier, self.pop_window,
                              self.num_threads]:
             self.check_if_positive(variable)

        if self.temp <= self.final_temp: raise ValueError("temp should be
            higher than final_temp")
        if self.tol <= self.final_tol: raise ValueError("tol should be
            higher than final_tol")
        #if self.num_recoms <= self.num_maps: raise ValueError("num_recoms
            should be larger than num_maps")
        if self.pop_window > 1: raise ValueError("pop_window should be
            between 0 and 1.")




    def check_if_int(self, variable):
        """Checks if the variable is an integer"""
        if isinstance(variable, int): pass
        elif isinstance(variable, float):
            if variable.is_integer() == True: pass
            else: raise ValueError(f"{variable} is not an integer")

    def check_if_positive(self, variable):
        """Checks if the variable is positive"""
        if variable <= 0: raise ValueError(f"{variable} is not positive")




class objective_vals:

    def __init__(self, ip):
        self.dev_vals = [0] * ip.total_obj_vals_entries
        self.avg_comp_vals = [0] * ip.total_obj_vals_entries
        self.eg_score_vals = [0] * ip.total_obj_vals_entries
        self.CDI_Count_vals = [0] * ip.total_obj_vals_entries
        self.excess_GU_vals = [0] * ip.total_obj_vals_entries
        self.mm_vals = [0] * ip.total_obj_vals_entries
        self.change_type_vals = [None] * ip.total_obj_vals_entries
        self.sa_probability = [None] * ip.total_obj_vals_entries
        self.map_conclusion_vals = [None] * (ip.total_iterations+1) #What
            happens to the map? Is it added to the archive? Is it discarded?
        self.best_pop = [0] * (ip.total_iterations+1) #Best population
            deviation in the archive at a given iteration
        self.best_comp = [0] * (ip.total_iterations+1) #Best compactness in
            the archive at a given iteration
        self.best_eg = [0] * (ip.total_iterations+1) #Best efficiency gap
            in the archive at a given iteration
        self.best_cdi = [0] * (ip.total_iterations+1) #Best CDI count in
            the archive at a given iteration
        self.best_excess_GU = [0] * (ip.total_iterations+1) #Best excess GU
            count in the archive at a given iteration
        self.best_mm = [0] * (ip.total_iterations+1) #Best median mean in
            the archive at a given iteration
        self.hvol = [0] * (ip.total_iterations+1) #Hypervolume at each
            iteration

    def fill_obj_vals(self, ip, dist_list, cdi_data, change, it,
        sa_prob=None):
        self.dev_vals[it] = District.pop_deviation(dist_list)
        self.avg_comp_vals[it] = District.comp_score(dist_list)
        self.eg_score_vals[it] = District.eg_score(dist_list)
        self.CDI_Count_vals[it] = cdi_data.cdi_count
        self.excess_GU_vals[it] = cdi_data.excess_GU
        self.mm_vals[it] = District.median_mean(dist_list)
        self.change_type_vals[it] = change
        self.sa_probability[it] = sa_prob
        #self.map_conclusion_vals[it] = None  ###NEED TO FIGURE OUT HOW TO
            DOCUMENT WHEN THE LAST FLIP OCCURS

    def __repr__(self):
        last_it = np.max(np.nonzero(self.dev_vals))
        dev = self.dev_vals[last_it]
        eg = self.eg_score_vals[last_it]
        cdi = self.CDI_Count_vals[last_it]
        eGU = self.excess_GU_vals[last_it]
        ct = self.change_type_vals[last_it]
        mm = self.mm_vals[last_it]
        return f"{last_it} iterations filled. Last entry: deviation =
            {dev}, eg = {eg}, cdi_count = {cdi}, excess_GU = {eGU}, mm =
            {mm}, change_type = {ct}"


class District:
    """A class that will hold the statistics associated with each
        district"""
    ideal_pop = None  #The ideal population that a district would like to be
    num_dists = None  #The number of districts

    def __init__(self, num):
        self.num = num  #District numbers will range from 0 to n-1
        self.Area = None  #Area in units of square kilometers
        self.Perimeter = None  #Perimeter in units of kilometers
        self.VoteCountRed = None  #Red votes in a district
        self.VoteCountBlue = None  #Blue votes in a district
        self.Population = None  #Population of the district
    def __eq__(self, other):
        """Tests whether other is equivalent to self"""
        return self.__dict__ == other.__dict__

    def __repr__(self):
        return f"District Number {self.num}. Population: {self.Population}.
            Area: {self.Area} km^2. Perimeter: {self.Perimeter} km. PP
            Score: {self.PPCompactScore}. Ideal population target:
            {self.ideal_pop}."

    def Nodes(self, G):
        '''Returns all nodes in this district, using graph G as input'''
        node_list = []
        for n in G.nodes():
            if G.nodes[n]["District Number"] == self.num:
                node_list.append(n)
        return node_list

    def BoundaryEdges(self, G):
        '''Returns all boundary edges for this district'''
        return list(nx.edge_boundary(G, self.Nodes(G)))

    def BoundaryNodes(self, G):
        '''Returns all boundary nodes for this district'''
        return list(nx.node_boundary(G, self.Nodes(G)))

    def Dist_nbrs(self, G):
        '''Returns all district neighbors'''
        Dist_Edge_List = set()
        for e in self.BoundaryEdges(G):
            e0_dist = G.nodes[e[0]]["District Number"]
            e1_dist = G.nodes[e[1]]["District Number"]
            if e0_dist < 0 or e1_dist < 0: ValueError("e0dist and e1_dist
                must be nonnegative.")
            Dist_Edge_List.add(e0_dist)
            Dist_Edge_List.add(e1_dist)
        Dist_Edge_List = list(Dist_Edge_List)
        try: Dist_Edge_List.remove(self.num)
        except ValueError: pass
        return Dist_Edge_List

    @property
    def PPCompactScore(self):
        '''Polsby-Popper Compactness Score (ranges from 0 to 1; 1 is
            best)'''
        try:
            if self.Perimeter < 0:
                raise ValueError("Perimeter must be nonnegative.")
            return 4 * math.pi * self.Area / self.Perimeter ** 2 if
                self.Perimeter != 0 else None
        except TypeError:
            return None

    @property
    def invPPCompactScore(self):
        '''Returns the inverse of the Polsby-Popper Compactness Score and
           subtracts 1 so that the ideal score is 0'''
        return (1 / self.PPCompactScore) - 1 if self.PPCompactScore != 0
           and self.PPCompactScore != None else None

    @property
    def shiftedPPCompactScore(self):
        '''Returns 1-PP. Range: [0,1], with 0 ideal. '''
        return 1 - self.PPCompactScore

    @property
    def TotalVotes(self):
        '''Sums red votes and blue votes for a district'''
        try:
            return self.VoteCountBlue + self.VoteCountRed
        except TypeError:
            return None

    @property
    def BlueShare(self):
        '''Returns the share of blue votes as a proportion of total votes'''
        try:
            return self.VoteCountBlue / self.TotalVotes if self.TotalVotes
               != 0 else None
        except TypeError:
            return None

    @property
    def EfficiencyGap(self):
        '''Returns the Efficiency gap as calculated by (wastedRed votes -
           wastedBlue votes) / total votes'''
        try:
            return (self.WastedRed - self.WastedBlue) / self.TotalVotes if
               self.TotalVotes != 0 else None
        except TypeError:
            return None

    @property
    def AbsEfficiencyGap(self):
        '''Returns the absolute value of the efficiency gap'''
        try:
            return abs(self.EfficiencyGap)
        except TypeError:
            return None

    @property
    def WinThreshold(self):
        '''Returns the number of votes needed to win an election in a
           district'''
        try:
            return math.ceil(self.TotalVotes/2 + 0.5)  #The '+0.5' is
               needed to deal with cases where the total number of votes
               is even
        except TypeError:
            return None
    @property
    def WastedBlue(self):
        '''Returns the number of wasted blue votes. If the blue party wins,
           then this value will be
        the number of blue votes beyond the win threshold. If the blue
           party loses, then this will be the
        number of blue votes. For coding simplicity, all ties are won by
           the blue party.'''
        try:
            if self.VoteCountBlue >= self.VoteCountRed:
                return self.VoteCountBlue - self.WinThreshold
            else:
                return self.VoteCountBlue
        except TypeError:
            return None

    @property
    def WastedRed(self):
        '''Returns the number of wasted red votes. If the red party wins,
           then this value will be
        the number of red votes beyond the win threshold. If the red party
           loses, then this will be the
        number of red votes. For coding simplicity, all ties are won by the
           blue party.'''
        try:
            if self.VoteCountRed >= self.VoteCountBlue:
                return self.VoteCountRed - self.WinThreshold
            else:
                return self.VoteCountRed
        except TypeError:
            return None

    @staticmethod
    def pop_list(dist_list):
        '''Returns a list of populations for each district'''
        pop_list = [d.Population for d in dist_list]
        pop_list = remove_nones(pop_list)
        return pop_list

    @staticmethod
    def EG_list(dist_list, AV=False):
        '''Returns a list of efficiency gaps for each district. If AV is
           true, then we return the absolute values'''
        if AV == False:
            eg_list = [d.EfficiencyGap for d in dist_list]
        else:
            eg_list = [d.AbsEfficiencyGap for d in dist_list]

        eg_list = remove_nones(eg_list)
        return eg_list

    @staticmethod
    def blue_share_list(dist_list):
        '''Returns a list of blue proportions of votes (as a fraction of
           total votes in district) for each district.'''
        bs_list = [d.BlueShare for d in dist_list]
        bs_list = remove_nones(bs_list)
        return bs_list


    def reset_vals(self):
        '''Resets several values. Useful during recom'''
        self.Area = 0
        self.Perimeter = 0
        self.VoteCountRed = 0
        self.VoteCountBlue = 0
        self.Population = 0


    def check_value_integrity(self):
        """Checks that all attributes make sense for the given district."""
        if self.Area < 0 or self.Population < 0 or self.Perimeter < 0 or
           self.VoteCountRed < 0 or self.VoteCountBlue < 0:
            raise ValueError("A metric in this district is negative.")

    @staticmethod
    def pop_deviation(dist_list):
        '''Returns a single positive integer that sums each district's
           deviation from the ideal population.
        Lower numbers for 'deviation' are better. A value of zero would
           indicate that every district has an equal number of people'''
        distcount = District.num_dists
        if distcount <= 0: raise ValueError("distcount should be positive")
        absdev = [0 for _ in range(distcount)]

        for i in range(distcount):
            if dist_list[i].Population != None:
                absdev[i] = abs(dist_list[i].Population -
                    District.ideal_pop)
        deviation = round(sum(absdev))
        return deviation

    @staticmethod
    def comp_score(dist_list, inverse=True):
        '''Computes the average Polsby-Popper score for a list of
           districts'''

        if inverse == False:
            comp_list = [d.PPCompactScore for d in dist_list]
        else:
            comp_list = [d.invPPCompactScore for d in dist_list]

        comp_list = remove_nones(comp_list)

        if len(comp_list) != District.num_dists:
            raise ValueError(f"comp_score did not return the proper number
                of districts. len(comp_list) = {len(comp_list)},
                District.num_dists = {District.num_dists}")
        PP_Comp_Score = sum(comp_list) / len(comp_list)  #Averages the
            compactness scores
        return PP_Comp_Score

    @staticmethod
    def eg_score(dist_list, AV=False):
        '''Computes the average efficiency gap score for the map'''
        eg_list = District.EG_list(dist_list, AV)  #Gets list of efficiency
            gaps
        eg_list = remove_nones(eg_list)

        if len(eg_list) != District.num_dists:
            raise ValueError(f"eg_score did not return the proper number of
                districts. len(eg_list) = {len(eg_list)},
                District.num_dists = {District.num_dists}")
        eg_score = abs(sum(eg_list) / len(eg_list))  #Averages the EG
            score, then takes absolute values
        return eg_score

    @staticmethod
    def median_mean(dist_list):
        """Returns the blue vote median-mean score for the map.
        Positive numbers are considered beneficial for the blue party.
        We return the absolute value of the median-mean score,
        and want values of zero."""
        bs_list = District.blue_share_list(dist_list)
        bs_median = statistics.median(bs_list)
        bs_mean = sum(bs_list) / len(bs_list)
        bmm_score = bs_median - bs_mean
        return abs(bmm_score)


class CDI:
    """A class that will contain all information about the
        county-district-intersection matrix"""
    def __init__(self, stateG):
        distcount = len(set(dict(stateG.nodes("District
            Number")).values()))  #Finds number of unique districts
        num_counties = len(set(dict(stateG.nodes("Reduced
            County")).values()))  #Finds number of unique counties
        units_in_CDI = np.zeros([distcount, num_counties], dtype=int)

        if distcount <= 0: raise ValueError("distcount is either 0 or
            negative.")
        if num_counties <= 0: raise ValueError("num_counties is either 0 or
            negative.")

        #Adds 1 to the matrix element A[i,j] if there is a GU in the ith
            district and (j-1)th county
        for n in stateG:
            i = stateG.nodes[n]["District Number"]
            j = stateG.nodes[n]["Reduced County"] - 1
            units_in_CDI[i][j] += 1

        self.cdi_mat = units_in_CDI
        self.distcount = distcount
        self.num_counties = num_counties

    @property
    def cdi_count(self):
        '''Counts number of nonzero entries in the CDI matrix. Then
            subtracts either the distcount
        or number of counties, so that the ideal value will be zero.'''
        return np.count_nonzero(self.cdi_mat) - max(self.distcount,
            self.num_counties)

    @property
    def excess_GU(self):
        '''GU stands for Geographical Unit. In this loop, we count the
            number of GUs in each county that
        are not in the most prevalent district. The ideal number of excess
            GUs is zero.'''
        excess_GU_mat = [0] * max(self.distcount, self.num_counties)
        transpose = self.cdi_mat.transpose()
        idx = 0
        for row in transpose:  #for each county
            maxval = max(row)  #maxval is the number of GUs in the county
                that belong to the most prominent district in that county
            excess_GU_mat[idx] = sum(row) - maxval
            idx += 1
        return sum(excess_GU_mat)

    def upd_cdi_mat_flip(self, stateG, GU, leaving_dist, entering_dist):
        '''Updates the cdi matrix after a flip'''
        if leaving_dist < 0 or entering_dist < 0:
            raise ValueError(f"Neither leaving_dist nor entering_dist
                should be negative. l_d = {leaving_dist}. e_d =
                {entering_dist}")
        self.cdi_mat[leaving_dist][stateG.nodes[GU]["Reduced County"] - 1]
            -= 1
        if self.cdi_mat[leaving_dist][stateG.nodes[GU]["Reduced County"] -
            1] < 0:
            raise ValueError(f"Entries in the CDI matrix cannot be
                negative. The CDI matrix is {self.cdi_mat}. GU = {GU}. l_d
                = {leaving_dist}, e_d = {entering_dist}.")
        self.cdi_mat[entering_dist][stateG.nodes[GU]["Reduced County"] - 1]
            += 1


class counters:
    '''A class that will contain the counters used in the simulated
        annealing step'''
    def __init__(self):
        self.flipcount = 0  #The number of flips done in total in the code
        self.recomcount = 0  #The number of recombination steps done in
            total in the code
        self.total_it_count = 0  #The number of iterations the code has done
        self.failed_recom_counter = 0  #The number of consecutive failed
            recombination steps in the current iteration
        self.failed_flip_counter = 0  #The number of consecutive failed
            flip steps in the current flip attempt
        self.alphacount = 0  #The number of alpha values utilized previously
        self.its_at_temp = 0  #The number of iterations that have occurred
            at this temperature

    @property
    def currentit(self):
        '''Returns the sum of the flip count and the recom count'''
        return self.flipcount + self.recomcount  #This returns the index
            that will be populated in obj_vals

    def __repr__(self):
        return f"flipcount = {self.flipcount}. recomcount =
            {self.recomcount}. failed_recom_counter =
            {self.failed_recom_counter}. currentit = {self.currentit}"


class Map_class:
    """A class that will contain a graph and its associated metric values"""
    def __init__(self, graph, alpha, dist_list, cdi_data):
        self.graph = graph
        self.alpha = alpha
        self.dist_list = dist_list
        self.cdi_data = cdi_data
        self.alpha.assigned = True

    def __eq__(self, other) :
        if isinstance(other, Map_class) == False: return False
        return self.__dict__ == other.__dict__

    @property
    def pop_dev(self):
        """Calculates population deviation for this map"""
        return District.pop_deviation(self.dist_list)

    @property
    def compactness(self):
        """Calculates Inverse Polsby-Popper compactness (minus 1) for this
            map"""
        return District.comp_score(self.dist_list)

    @property
    def eg(self):
        """Calculates efficiency gap score for this map"""
        return District.eg_score(self.dist_list)

    @property
    def cdi_num(self):
        """Calculates the number of county-district intersections for this
            map"""
        return self.cdi_data.cdi_count

    @property
    def excess_GU_num(self):
        """Calculates the number of excess GUs for this map"""
        return self.cdi_data.excess_GU

    @property
    def mm(self):
        """Calculates the median-mean score for the map"""
        return District.median_mean(self.dist_list)

    @property
    def metrics(self):
        """Returns a list of metrics"""
        return [self.pop_dev, self.compactness, self.eg, self.cdi_num,
            self.excess_GU_num, self.mm]


    def compare_objs(self, rng, pareto_set, ip):
        """Compares the current map with all maps in the pareto set to
            determine if
        1. The current map dominates at least one map in the pareto set
        2. The current map is dominated by at least one map in the pareto
            set
        3. The current map neither dominates nor is dominated by any map in
            the pareto set
        """
        nonequal_flag = False #Flag that is True if some map in the PS is
            not equal to self
        rng.shuffle(pareto_set)
        dominated_maps = []  #Contains the set of maps that are dominated
            by self
        for Map in pareto_set:
            objs_to_check = [(self.pop_dev, Map.pop_dev),
                             (self.compactness, Map.compactness),
                             (self.eg, Map.eg),
                             (self.cdi_num, Map.cdi_num),
                             (self.excess_GU_num, Map.excess_GU_num),
                             (self.mm, Map.mm)]
            all_objs = deepcopy(objs_to_check)
            objs_to_check = [objs_to_check[i] for i in
                range(len(ip.metrics_to_use)) if ip.metrics_to_use[i] == 1]
            diff = []
            for entry in objs_to_check:
                diff.append(int(np.sign(entry[1] - entry[0])))  #old-new
                    sign #PS - candidate

            if list(set(diff)) == [0]:
                #This is the case where all metrics are equal. May occur if
                    we compare self to self
                continue
            elif list(set(diff)) == [-1]:
                #This is the case where self (candidate) is dominated
                if dominated_maps != []: raise ValueError("This map was
                    both dominant and dominated. This shouldn't happen")
                return -1, Map
            elif list(set(diff)) == [1]:
                #This is the case where self dominates
                dominated_maps.append(Map)
                nonequal_flag = True
                #return 1, Map
            else:
                #This is the case where self neither dominates nor is
                    dominated.
                #We need to check all maps, so we pass here
                nonequal_flag = True  #signifies that at least one
                    different map is in the Pareto set

        if nonequal_flag == False: #We will be here if list(set(diff)) ==
            [0] for every Map in pareto_set
            all_objs_diffs = []
            for tuple_ in all_objs:
                all_objs_diffs.append(tuple_[1] - tuple_[0])
            if list(set(all_objs_diffs)) == [0]:
                return -2, -2  #Returns -2, -2 if the maps in the Pareto
                    set all are exactly the same as self (this shouldn't
                    happen)
            else: return 0, 0
        elif dominated_maps != []:
            return 1, dominated_maps
        else:
            return 0, 0  #Returns 0, 0 if self is not dominated by any map
                and self does not dominate any map

    def check_objs(self, ip):
        """Checks whether the objectives of this plan are under the
            user-defined upper bounds"""
        under_ub = []
        for idx, met_used in enumerate(ip.metrics_to_use):
            if met_used == 1:
                under_ub.append(self.metrics[idx]<=ip.met_maxes[idx])
            #else: #met_used==0
                #under_ub.append(None)
        return under_ub


class Weight_vector:
    """A member of this class is a weight vector alpha, where the entries
        sum to 1. """
    def __repr__(self):
        return f"{self.alpha.wv}"



    def __init__(self, rng, metric_count):
        self.assigned = False
        alpha = []
        for _ in range(metric_count):
            alpha.append(rng.randint(1, 100000))
        alpha = norm(alpha)
        self.wv = alpha

def hypervolume(list_of_maps, ip):
    """Calculates the hypervolume given a list of maps"""
    ref = [1.1 for _ in range(sum(ip.metrics_to_use))]
    ind = HV(ref_point=ref)
    metrics = [None for _ in range(len(list_of_maps))]
    for idx, map_ in enumerate(list_of_maps):
        for i in range(len(ip.metrics_to_use)):
            if map_.metrics[i] > ip.met_maxes[i] and ip.metrics_to_use[i]
                == 1:
                raise RuntimeError(f"We shouldn't have the objective values
                    exceed the met_max. map_.metrics = {map_.metrics},
                    ip.met_maxes = {ip.met_maxes}")
        metrics[idx] = [map_.metrics[i]/ip.met_maxes[i] for i in
            range(len(ip.metrics_to_use)) if ip.metrics_to_use[i] == 1]
    metrics = np.array(metrics)
    volume = ind(metrics)
    return volume

def remove_nones(val_list):
    """Removes all 'None' entries from a list"""
    return [val for val in val_list if val != None]

def norm(vector):
    """Input is a list of nonnegative numbers. Returns a normed vector,
     i.e. a vector that sums to 1."""
    if any(t < 0 for t in vector):
        raise ValueError("All entries in this vector must be nonnegative.")
    tot = sum(vector)
    for i in range(len(vector)):
        vector[i] = vector[i] / tot
    eps = 0.000001
    if sum(vector) > 1 + eps or sum(vector) < 1 - eps:
        raise ValueError("The elements of this vector must sum to 1.
            Something is wrong with the norm method.")
    return vector


def build_alpha(rng, metric_count, ip):
    '''Builds the normalized weight vectors for use in simulated
     annealing'''
    if len(ip.alphas) > ip.num_maps:
        raise ValueError("Too many weight vectors were chosen. We must have
            fewer than the number of archive spots.")
    for vec in ip.alphas:
        if len(vec) != 6:
            raise ValueError("Each weight vector must be a length of 6.")
        for idx, entry in enumerate(vec):
            if (ip.metrics_to_use[idx] == 0 and entry != 0) or
                (ip.metrics_to_use[idx] != 0 and entry == 0):
                raise ValueError("Each entry in the weight vectors should
                    match the metrics_to_use list")
    alphas = [None] * ip.num_maps
    if ip.preselect_alphas == False:
        for i in range(ip.num_maps):
            alphas[i] = Weight_vector(rng, metric_count)
    else: #If we want specific weight vectors
```

```
        for i in range(ip.num_maps):
            alphas[i] = Weight_vector(rng, metric_count)
        i=0
        for vec in ip.alphas:
            for j in range(len(vec)):
                if vec[j] == 0: vec[j] = None
            vec = remove_nones(vec)
            alphas[i].wv = norm(vec)

            i+=1

    return alphas


def initialize_map3(rng, ip):
    '''Builds a graph based on a json file supplied by the user'''
    #stateG = Graph.from_file(ip.file_json, adjacency="rook",
        reproject="True", ignore_errors="False")
    with open('stateG.pkl', 'rb') as fp: stateG = pickle.load(fp)
    create_county_dict(stateG)
    stateG = generate_random_starting_map2(rng, stateG, ip)
    return stateG


def create_county_dict(stateG):
    '''Populates county_dict with 1-x, based on the sorted county numbers'''
    county_list = dict(stateG.nodes("COUNTY")).values()
    county_list = list(map(int, county_list))  #Converts strings to integers
    county_list = sorted(list(set(county_list)))  #Sorts the list and
        deletes duplicate values
    county_dict = {}
    i = 1
    for county in county_list:
        county_dict[county] = i  #Populates a dictionary that associates
            each original county value with its sorted value
        i += 1
    for n in stateG.nodes:
        stateG.nodes[n]["Reduced County"] =
            county_dict[int(stateG.nodes[n]["COUNTY"])]


def generate_random_starting_map2(rng, G, ip):
    '''Generates a random starting map for the graph of G by generating 7
        random centers'''
    tree = wilson(G, random)
    #spl = dict(nx.all_pairs_shortest_path_length(G))
    spl = dict(nx.all_pairs_shortest_path_length(tree))
    while True:
        #centers = rng.sample(list(G.nodes), ip.distcount)
        centers = rng.sample(list(tree.nodes), ip.distcount)
        dist_pops = [0]*ip.distcount
        #for n in list(G.nodes):
        for n in list(tree.nodes):
            n_len_to_center = [None]*ip.distcount
            for i in range(len(centers)):
                n_len_to_center[i] = spl[n][centers[i]]
            minlen = min(n_len_to_center)
            minlenidx = n_len_to_center.index(minlen)
            G.nodes[n]["District Number"] = minlenidx
            G.nodes[n]["Original District Number"] = minlenidx
            tree.nodes[n]["District Number"] = minlenidx
            tree.nodes[n]["Original District Number"] = minlenidx

            #Calculate the objectives
            dist_pops[minlenidx] += tree.nodes[n]["POPULATION"]

        District.ideal_pop = sum(dist_pops) / ip.distcount
        #if all(dp > District.ideal_pop*0.3 and dp < District.ideal_pop*3
            for dp in dist_pops):
        if min(dist_pops) > District.ideal_pop*0.4 and max(dist_pops) <
            District.ideal_pop*3:
            break
        else:
            print("Failed to create a proper partitioning. Retrying...")
            continue

    #Initializing Boundary status for each edge:
    dist_boundary = {}  #Initializes a dictionary that will describe each
        edge as being a boundary edge or not
    for e in G.edges:
        n0 = e[0]
        n1 = e[1]
        n0_dist = G.nodes[n0]["District Number"]
        n1_dist = G.nodes[n1]["District Number"]
        if n0_dist != n1_dist:
            if n0_dist < 0 or n1_dist < 0:
                raise ValueError("The district should not be negative")
            else:
                dist_boundary[e] = 1  #If edge represents a district
                    boundary
        else:
            dist_boundary[e] = 0  #If edge is not a district boundary

    print("Adding Boundary status attribute for edges")
    nx.set_edge_attributes(G, dist_boundary, "Dist_Boundary")

    return G


def populate_dist_list(stateG, dist_list):
    '''Populates dist_list with all statistics based on the stateG graph'''
    distcount = len(set(dict(stateG.nodes("District Number")).values()))
        #Finds number of unique districts
    for i in range(distcount):
        dist_list[i] = District(i)  #Reinitializes (and therefore resets)
            the dist_list so that district numbers range from 0 to n-1.

    for n in stateG.nodes:
        dist_num = stateG.nodes[n]["District Number"]
        if dist_list[dist_num].Area == None:
            dist_list[dist_num].Area = 0
        if dist_list[dist_num].Population == None:
            dist_list[dist_num].Population = 0
        if dist_list[dist_num].VoteCountBlue == None:
            dist_list[dist_num].VoteCountBlue = 0
        if dist_list[dist_num].VoteCountRed == None:
            dist_list[dist_num].VoteCountRed = 0

        dist_list[dist_num].Area += stateG.nodes[n]["area"]
        dist_list[dist_num].Population += stateG.nodes[n]["POPULATION"]
        dist_list[dist_num].VoteCountBlue += stateG.nodes[n]["PresBlue"]
        dist_list[dist_num].VoteCountRed += stateG.nodes[n]["PresRed"]

    ideal_pop = round(sum(District.pop_list(dist_list)) / distcount)
    District.ideal_pop = ideal_pop
    District.num_dists = distcount

    print(f"The starting population of each district is
        {District.pop_list(dist_list)}. Thus, the ideal population for a
        district is {ideal_pop}.")

    for d in dist_list:
        if d.Perimeter == None:
            d.Perimeter = 0

    for e in list(stateG.edges):  #Cycles through all edges to find
        district perimeter
        if stateG[e[0]][e[1]]["Dist_Boundary"] == 1:
            dist_num0 = stateG.nodes[e[0]]["District Number"]
            dist_num1 = stateG.nodes[e[1]]["District Number"]

            dist_list[dist_num0].Perimeter +=
                stateG[e[0]][e[1]]["shared_perim"]
            dist_list[dist_num1].Perimeter +=
                stateG[e[0]][e[1]]["shared_perim"]

    for n in stateG.nodes:  #Cycles through all nodes to find boundary
        nodes and add state edge perimeter
        if stateG.nodes[n]["boundary_node"] == True:
            dist_num = stateG.nodes[n]["District Number"]
            dist_list[dist_num].Perimeter +=
                stateG.nodes[n]["boundary_perim"]


def flip(rng, map_for_flip, temp, ip, count, dist1=None, dist2=None,
    procid=None):
    '''This is the flip algorithm. We move one GU across district lines.
    If GU and entering_dist are provided, then we consider that possible
        flip.
    If dist1 and dist2 are provided, we will flip one GU from dist1 to
        dist2 or vice-versa.'''
    stateG = map_for_flip.graph
    dist_list = map_for_flip.dist_list
    cdi_data = map_for_flip.cdi_data
    distcount = len(dist_list)
```

```
#Records starting values for these inputs
s_stateG = deepcopy(stateG)
s_dist_list = deepcopy(dist_list)
s_cdi_data = deepcopy(cdi_data)

adj_flag = False
edge_count = stateG.number_of_edges()

if (dist1 == None and dist2 != None) or (dist1 != None and dist2 ==
    None):
    ValueError("dist1 and dist2 must both be None or both be inputs")
if dist1 == dist2:
    ValueError(f"dist1 and dist2 cannot be equal. They are both
        {dist1}")
if dist1 != None and (dist1 >= distcount or dist1 < 0):
    raise ValueError("dist1 was outside of the appropriate range.")
if dist2 != None and (dist2 >= distcount or dist2 < 0):
    raise ValueError("dist2 was outside of the appropriate range.")


while adj_flag == False:
    #This portion of code finds a random boundary edge
    boundaryflag = False

    while boundaryflag == False:
        if dist1 != None:  #Finds a boundary edge on the dist1-dist2
            border
            dist1_b_edges = dist_list[dist1].BoundaryEdges(stateG)
            dist2_b_edges = dist_list[dist2].BoundaryEdges(stateG)
            for idx, e in enumerate(dist1_b_edges):
                if e[0] > e[1]:
                    dist1_b_edges[idx] = (e[1], e[0])
            for idx, e in enumerate(dist2_b_edges):
                if e[0] > e[1]:
                    dist2_b_edges[idx] = (e[1], e[0])
            common_edges = list(set(dist1_b_edges) &
                set(dist2_b_edges))  #Finds boundary edges that both
                dist1 and dist2 have
            if common_edges == []: raise RuntimeError("common_edges
                should not be empty")
            pair = list(rng.choice(common_edges))
            rng.shuffle(pair)
            GU = pair[0]
            other_GU = pair[1]
            boundaryflag = True
        else:  #Finds a random boundary edge
            rand_edge = rng.randint(0, edge_count - 1)  #Selects a
                random edge from stateG.edges
            pair = list(stateG.edges)[rand_edge]
            rng.shuffle(pair)
            GU = pair[0]
            other_GU = pair[1]
            if stateG[GU][other_GU]["Dist_Boundary"] == 1:  #If the
                edge represents a district boundary
                boundaryflag = True
            else:
                continue  #Resets the loop if the edge does not
                    represent a district boundary

    #This portion of code determines which GU will move
    leaving_dist = stateG.nodes[GU]["District Number"]
    entering_dist = stateG.nodes[other_GU]["District Number"]
    if leaving_dist == entering_dist:
        raise RuntimeError("leaving_dist should not be the same as
            entering_dist")

    DistrictNodes = dist_list[leaving_dist].Nodes(stateG)  #Finds all
        nodes in leaving_dist
    DistrictNodes.remove(GU)
    sg_for_leaving_dist = stateG.subgraph(DistrictNodes)  #Creates a
        subgraph containing all nodes from DistrictNodes
    if nx.is_connected(sg_for_leaving_dist):
        adj_flag = True  #True if the leaving_dist is still contiguous
    else:
        adj_flag = False  #Keeps the adj_flag at False if the move
            would create a discontiguity

stateG.nodes[GU]["District Number"] = entering_dist  #Changes the
    district number in stateG

for nbr in stateG.neighbors(GU):  #Cycles through neighboring nodes to
    adjust boundary status, perimeter, and district neighbors
    nbr_dist = stateG.nodes[nbr]["District Number"]
    if nbr_dist == entering_dist:
        stateG[GU][nbr]["Dist_Boundary"] = 0
        dist_list[entering_dist].Perimeter -=
            stateG[GU][nbr]["shared_perim"]  #This is now 'interior'
            perimeter
        dist_list[leaving_dist].Perimeter -=
            stateG[GU][nbr]["shared_perim"]  #Neither GU nor nbr is in
            leaving_dist

    elif nbr_dist == leaving_dist:  #If nbr is part of the leaving_dist
        stateG[GU][nbr]["Dist_Boundary"] = 1
        dist_list[entering_dist].Perimeter +=
            stateG[GU][nbr]["shared_perim"]
        dist_list[leaving_dist].Perimeter +=
            stateG[GU][nbr]["shared_perim"]

    else:  #Neighboring district is not entering_dist, leaving_dist, or
        the dummy district
        stateG[GU][nbr]["Dist_Boundary"] = 1
        dist_list[entering_dist].Perimeter +=
            stateG[GU][nbr]["shared_perim"]
        dist_list[leaving_dist].Perimeter -=
            stateG[GU][nbr]["shared_perim"]

dist_list[entering_dist].Population += stateG.nodes[GU]["POPULATION"]
    #Adds population to the dist_list population entry corresponding to
    entering_dist
    dist_list[leaving_dist].Population -= stateG.nodes[GU]["POPULATION"]
        #Subtracts population from the dist_list population entry
        corresponding to leaving_dist

    dist_list[entering_dist].Area += stateG.nodes[GU]["area"]  #Adds area
        to the dist_list area entry corresponding to entering_dist
    dist_list[leaving_dist].Area -= stateG.nodes[GU]["area"]  #Subtracts
        area from the dist_list area entry corresponding to leaving_dist

    dist_list[entering_dist].VoteCountRed += stateG.nodes[GU]["PresRed"]
        #Adds red votes to the dist_list red votes entry corresponding to
        entering_dist
    dist_list[leaving_dist].VoteCountRed -= stateG.nodes[GU]["PresRed"]
        #Subtracts red votes from the dist_list red votes entry
        corresponding to leaving_dist

    dist_list[entering_dist].VoteCountBlue += stateG.nodes[GU]["PresBlue"]
        #Adds blue votes to the dist_list blue votes entry corresponding to
        entering_dist
    dist_list[leaving_dist].VoteCountBlue -= stateG.nodes[GU]["PresBlue"]
        #Subtracts blue votes from the dist_list blue votes entry
        corresponding to leaving_dist

    cdi_data.upd_cdi_mat_flip(stateG, GU, leaving_dist, entering_dist)
        #Updates the CDI matrix

    map_u = Map_class(s_stateG, map_for_flip.alpha, s_dist_list,
        s_cdi_data)  #old
    map_v = Map_class(stateG, map_for_flip.alpha, dist_list, cdi_data)
        #new, proposed


    if temp == None: return True
    do_flip, sa_prob = sa_prob_calc(rng, map_v, map_u, temp, ip, count,
        type_="Flip")
    #print(f"PI{procid}. R{count.recomcount}. Completed Flip algorithm.
        Flipped GU {GU} from district {leaving_dist} to district
        {entering_dist}.")
    if leaving_dist == entering_dist:
        print("entering_dist and leaving_dist are the same. This should not
            happen.")
    return do_flip, sa_prob  #Will return True if the flip should be done,
        False otherwise


def recom(rng, map_v, tol, count, dist1=None, dist2=None, procid=None):
    '''Does a Recombination step for the graph stateG
    1. Determine if two districts are adjacent
    2. Grab all GUs from those two districts and create a subgraph.
    3. Error check: Verify that the subgraph is connected
    4. Wilson's Algorithm
    5. Make sure that the resulting tree acquires all attributes from stateG
    6. FindEdgeCut
    7. Reassign the new subgraphs to their proper districts'''
    stateG = map_v.graph
```

```
dist_list = map_v.dist_list
#distcount = len(set(dict(stateG.nodes("District Number")).values()))
    #Finds number of unique districts
distcount = District.num_dists
dist_adj_flag = False
if dist1 == None and dist2 != None:
    raise ValueError("Either both dist1 and dist2 must be None or
        neither should be")
if dist2 == None and dist1 != None:
    raise ValueError("Either both dist1 and dist2 must be None or
        neither should be")
if dist1 != None and (dist1 >= distcount or dist1 < 0):
    raise ValueError("dist1 was outside of the appropriate range.")
if dist2 != None and (dist2 >= distcount or dist2 < 0):
    raise ValueError("dist2 was outside of the appropriate range.")

while dist_adj_flag == False:
    if dist1 == None:
        dist1 = rng.randint(0, distcount - 1)  #Randomly selects dist1
            if it wasn't provided as input or if it is out of range
        while dist2 == None:
            dist2 = rng.choice(dist_list[dist1].Dist_nbrs(stateG))
                #Finds an appopriate dist2
    if dist1 not in dist_list[dist2].Dist_nbrs(stateG):
        dist1 = None
        dist2 = None
        continue  #Restarts the loop if this district neighbor pair
            can't be located in dist_list
    d1pop = dist_list[dist1].Population
    d2pop = dist_list[dist2].Population
    if np.sign(d1pop - District.ideal_pop) == np.sign(d2pop -
        District.ideal_pop):
        dist1 = None
        dist2 = None
        continue  #Restarts the loop if this district neighbor pair
            doesn't have pops on opposite sides of ideal_pop

    dist1_and_dist2_nodes = []
    for n in stateG.nodes():
        if stateG.nodes[n]["District Number"] == dist1 or
            stateG.nodes[n]["District Number"] == dist2:
            dist1_and_dist2_nodes.append(n)
    two_dist_graph = stateG.subgraph(dist1_and_dist2_nodes)  #Creates a
        subgraph of the two districts
    if nx.is_connected(two_dist_graph) == False:
        print(f"District {dist1} and District {dist2} are not adjacent.
            Reselecting districts")
        dist1 = None
        dist2 = None
        dist_adj_flag = False  #Keeps the flag at False

    else:  #If the two districts are indeed adjacent
        dist_adj_flag = True
```

```
tree = wilson(two_dist_graph, rng)  #Creates a uniform random spanning
    tree for two_dist_graph using Wilson's algorithm
subgraphs = find_edge_cut(rng, tree, tol)  #Finds an edge to remove
    from the tree to create two districts

#This next section of code decides which subgraph should become
    district 1 and which should become district 2
if subgraphs:  #If subgraphs is not empty
    s0d1count = 0
    s0d2count = 0
    s1d1count = 0
    s1d2count = 0
    for i in subgraphs[0]:
        if stateG.nodes[i]["District Number"] == dist1:
            s0d1count += 1
        elif stateG.nodes[i]["District Number"] == dist2:
            s0d2count += 1
    for i in subgraphs[1]:
        if stateG.nodes[i]["District Number"] == dist1:
            s1d1count += 1
        elif stateG.nodes[i]["District Number"] == dist2:
            s1d2count += 1

    #Assigns either dist1 or dist2 to the moved GUs
    if s0d1count + s1d2count >= s0d2count + s1d1count:
        for i in subgraphs[0]:
            stateG.nodes[i]["District Number"] = dist1
        for i in subgraphs[1]:
            stateG.nodes[i]["District Number"] = dist2

    else:
        for i in subgraphs[0]:
            stateG.nodes[i]["District Number"] = dist2
        for i in subgraphs[1]:
            stateG.nodes[i]["District Number"] = dist1

    #Resets dist_list entries
    dist_list[dist1].reset_vals()
    dist_list[dist2].reset_vals()

    #Cycles through nodes to update boundary list
    for GU in two_dist_graph.nodes:
        GU_dist = stateG.nodes[GU]["District Number"]
        for nbr in list(stateG.neighbors(GU)):  #Cycles through
           neighboring nodes to update boundary status, perimeter, and
           Dist_nbrs
            nbr_dist = stateG.nodes[nbr]["District Number"]

            if nbr_dist == GU_dist:
                stateG[GU][nbr]["Dist_Boundary"] = 0
                dist_list[GU_dist].Perimeter += 0

            else:  #If the GU and nbr are in different districts
                stateG[GU][nbr]["Dist_Boundary"] = 1
```

```
                    dist_list[GU_dist].Perimeter +=
                        stateG[GU][nbr]["shared_perim"]

            #Cycles through all nodes to find boundary nodes and add state
                edge perimeter
            if stateG.nodes[GU]["boundary_node"] == True:
                dist_num = stateG.nodes[GU]["District Number"]
                dist_list[dist_num].Perimeter +=
                    stateG.nodes[GU]["boundary_perim"]

            #Updates the dist_list instance
            dist_list[GU_dist].Area += stateG.nodes[GU]["area"]
            dist_list[GU_dist].Population += stateG.nodes[GU]["POPULATION"]
            dist_list[GU_dist].VoteCountRed += stateG.nodes[GU]["PresRed"]
            dist_list[GU_dist].VoteCountBlue += stateG.nodes[GU]["PresBlue"]

            dist_list[GU_dist].check_value_integrity()  #Returns an error
                if anything is negative

        map_v.cdi_data = CDI(stateG)  #Reinitializes the CDI data after
            this recom step.

        print(f"PI{procid}. Recom number {count.recomcount} succeeded.
            Reorganized districts {dist1} and {dist2}")

        return dist1, dist2, True  #Indicates that the recom_success_flag
            is True
    else:  #If subgraphs were empty (i.e. we couldn't find an edge to cut
        that split population within tolerance)
        map_v.cdi_data = CDI(stateG)
        return dist1, dist2, False  #Indicates that the recom_success_flag
            is False


def wilson(graph, rng):
    '''Returns a uniform spanning tree on G'''
    walk = loopErasedWalk(graph, rng)
    currentNodes = [n for n in walk]

    uniformTree = nx.Graph()
    for i in range(len(walk) - 1):
        uniformTree.add_edge(walk[i], walk[i + 1])

    treeNodes = set(uniformTree.nodes)
    neededNodes = set(graph.nodes) - treeNodes

    while neededNodes:
        v = rng.choice(sorted(list(neededNodes))) # sort for code
            repeatability
        walk = loopErasedWalk(graph, rng, v1 = [v], v2 = currentNodes)
        currentNodes += walk
        for i in range(len(walk) - 1):
            uniformTree.add_edge(walk[i], walk[i + 1])
        treeNodes = set(uniformTree.nodes)
        neededNodes = set(graph.nodes) - treeNodes
    pass
    nx.set_node_attributes(uniformTree, dict(graph.nodes("POPULATION")),
        "POPULATION")
    nx.set_node_attributes(uniformTree, dict(graph.nodes("District
        Number")), "District Number")
    nx.set_node_attributes(uniformTree, dict(graph.nodes("Original District
        Number")), "Original District Number")
    nx.set_node_attributes(uniformTree, dict(graph.nodes("Reduced
        County")), "Reduced County")
    nx.set_node_attributes(uniformTree, dict(graph.nodes("area")), "area")
    nx.set_node_attributes(uniformTree, dict(graph.nodes("PresBlue")),
        "PresBlue")
    nx.set_node_attributes(uniformTree, dict(graph.nodes("PresRed")),
        "PresRed")
    nx.set_edge_attributes(uniformTree, dict(graph.edges("shared_perim")),
        "shared_perim")
    nx.set_edge_attributes(uniformTree, dict(graph.edges("Dist_Boundary")),
        "Dist_Boundary")
    return uniformTree


def loopErasedWalk(graph, rng, v1 = None, v2 = None):
    '''Returns a loop-erased random walk between components v1 & v2'''
    if v1 is None:
        v1 = [rng.choice(sorted(list(graph.nodes)))]
    if v2 is None:
        v2 = [rng.choice(sorted(list(graph.nodes)))]

    v = rng.choice(sorted(v1))
    walk = [v]
    while v not in v2:
        v = rng.choice(sorted(list(graph.neighbors(v))))
        if v in walk:
            walk = walk[0:walk.index(v)]
        walk.append(v)

    return walk


def find_edge_cut(rng, tree, tol):
    '''Input a tree graph and a percent tolerance. The function will remove
        a
    random edge that splits the tree into two pieces such that each piece
    has population within that percent tolerance. The variable 'tol' should
        be a positive real
    number in (0,100].'''
    if tol > 100 or tol <= 0 or (isinstance(tol, float) == False and
        isinstance(tol, int) == False):
        raise ValueError(f"tol={tol} must be a float or integer variable in
            the range (0,100].")
    if nx.is_tree(tree) == False:
        raise ValueError("The input graph must be a tree.")
    tree_edge_list = list(tree.edges)
    rng.shuffle(tree_edge_list)  #Randomly shuffles the edges of T
    e = None
    num_edges = len(tree_edge_list)

    for i in range(num_edges):
        e = tree_edge_list[i]  #Edge to delete
        tree.remove_edge(*e)
        subgraphs = nx.connected_components(tree)
        subgraphs_lst = list(subgraphs)
        subgraphs_lst[0] = sorted(subgraphs_lst[0])
        subgraphs_lst[1] = sorted(subgraphs_lst[1])
        dist_pop1 = sum(value for key, value in
            nx.get_node_attributes(tree, "POPULATION").items() if key in
            subgraphs_lst[0])  #Finds population sum for first district
        dist_pop2 = sum(value for key, value in
            nx.get_node_attributes(tree, "POPULATION").items() if key in
            subgraphs_lst[1])  #Finds population sum for second district
        total_pop = dist_pop1 + dist_pop2
        avg_pop = total_pop / 2
        if abs(dist_pop1 - avg_pop) > 0.01 * tol * avg_pop or abs(dist_pop2
            - avg_pop) > 0.01 * tol * avg_pop:  #If either proposed
            district is outside the prescribed tolerance
            tree.add_edge(*e)  #Adds the edge back to the tree if it didn't
                meet the tolerance
        else:  #This is what we want: both proposed districts within the
            prescribed tolerance
            # print(f"Population requirement was met. Removing edge {e}.
                Required {i+1} iteration.")
            # pass
            return subgraphs_lst

        if i == num_edges - 1:
            #print(f"No subgraphs with appropriate criteria requirements
                were found. Required {i+1} iterations.")
            return []  #Returns empty subgraphs list if no appropriate
                subgraphs were found.


def sa_prob_calc(rng, map_v, map_u, temp, ip, count, type_="Recom"):
    """Uses simulated annealing architecture to determine whether we should
        discard map_v or make it our current solution.
    Positive values for delta_f are preferred, since we are subtracting old
        solution minus new solution. """
    # pop_dev_diff =     map_u.pop_dev - map_v.pop_dev
    # compactness_diff = map_u.compactness - map_v.compactness
    # eg_diff =          map_u.eg - map_v.eg
    # cdi_diff =         map_u.cdi_num - map_v.cdi_num
    # eGU_diff =         map_u.excess_GU_num - map_v.excess_GU_num
    # mm_diff =          map_u.mm - map_v.mm

    met_diffs = [map_u.metrics[i] - map_v.metrics[i] for i in
        range(len(ip.metrics_to_use))]
    pop_dev_diff =     met_diffs[0]
    compactness_diff = met_diffs[1]
    eg_diff =          met_diffs[2]
    cdi_diff =         met_diffs[3]
```

```
eGU_diff =          met_diffs[4]
mm_diff =           met_diffs[5]

append_diffs(ip, type_, met_diffs)

if type_ == "Flip":
    objscales = ip.objective_scales_flips
elif type_ == "Recom":
    objscales = ip.objective_scales
else:
    raise ValueError(f"'type_' is not defined properly from
        sa_prob_calc. type_ = {type_}")

#These obj_diffs are weighted so that all objectives are of similar
    value
obj_diffs = [None] * len(ip.metrics_to_use)
distance_from_pop_window_u = [None] * ip.distcount
for i, pop in enumerate(District.pop_list(map_u.dist_list)):
    if pop > District.ideal_pop*(1+ip.pop_window):
        distance_from_pop_window_u[i] = pop -
            District.ideal_pop*(1+ip.pop_window) #population is too
            large
    elif District.ideal_pop*(1-ip.pop_window) <= pop <=
        District.ideal_pop*(1+ip.pop_window):
        distance_from_pop_window_u[i] = 0 #population is in the target
            window
    else:
        distance_from_pop_window_u[i] = pop -
            District.ideal_pop*(1-ip.pop_window) #population is too
            small

distance_from_pop_window_v = [None] * ip.distcount
for i, pop in enumerate(District.pop_list(map_v.dist_list)):
    if pop > District.ideal_pop*(1+ip.pop_window):
        distance_from_pop_window_v[i] = pop -
            District.ideal_pop*(1+ip.pop_window) #population is too
            large
    elif District.ideal_pop*(1-ip.pop_window) <= pop <=
        District.ideal_pop*(1+ip.pop_window):
        distance_from_pop_window_v[i] = 0 #population is in the target
            window
    else:
        distance_from_pop_window_v[i] = pop -
            District.ideal_pop*(1-ip.pop_window) #population is too
            small

if all(distance_from_pop_window_u) == 0 and
    all(distance_from_pop_window_v) == 0:
#if map_u.pop_dev < ip.pop_window*District.ideal_pop and  map_v.pop_dev
    < ip.pop_window*District.ideal_pop:
    #if both the old map and new map are within the acceptable
        population window, pop_multiplier is set to 1.
    obj_diffs[0] = pop_dev_diff / objscales[0] if ip.metrics_to_use[0]
        == 1 else None
else:
        #if either the new map or old map has a population outside the
            acceptable population window, emphasize pop with pop_multiplier
        obj_diffs[0] = pop_dev_diff * ip.pop_multiplier(count) /
            objscales[0] if ip.metrics_to_use[0] == 1 else None
    obj_diffs[1] = compactness_diff / objscales[1] if ip.metrics_to_use[1]
        == 1 else None
    obj_diffs[2] = eg_diff / objscales[2] if ip.metrics_to_use[2] == 1 else
        None
    obj_diffs[3] = cdi_diff / objscales[3] if ip.metrics_to_use[3] == 1
        else None
    obj_diffs[4] = eGU_diff / objscales[4] if ip.metrics_to_use[4] == 1
        else None
    obj_diffs[5] = mm_diff / objscales[5] if ip.metrics_to_use[5] == 1 else
        None

    obj_diffs = remove_nones(obj_diffs)

    #change in energy
    delta_f = np.dot(map_u.alpha.wv, obj_diffs)
    try:
        rho = math.exp(delta_f / temp) if delta_f < 0 else 1  #If delta_f <
            0, we got worse.
    except OverflowError:
        rho = 0
    if rho > 1 or rho < 0:
        raise ValueError(f"rho should be between 0 and 1. rho = {rho}")
    r = rng.uniform(0,1)
    if r <= rho:
        return True, rho
    else:
        return False, rho

def append_diffs(ip, type_, met_diffs):
    """For analysis. Appends an entry to examine the differences found in
        each iteration for each metric"""
    if type_ == "Flip":
        try: ip.met_diffs_flips.append(met_diffs)
        except AttributeError: ip.met_diffs_flips = []
    elif type_ == "Recom":
        try: ip.met_diffs_recoms.append(met_diffs)
        except AttributeError: ip.met_diffs_recoms = []
    return


def replace_map(rng, pareto_set, map_p, map_v):
    """Replaces map_p in the Pareto set with map_v"""

    if isinstance(map_p, list) == False:
        map_v.alpha = map_p.alpha
        map_p = [map_p]  #If only one map is in map_p, we make it into a
            list
    else:
        map_v.alpha = rng.choice(map_p).alpha
    for map in map_p:
        pareto_set.remove(map)
        map.alpha.assigned = False
    pareto_set.append(map_v)
    map_v.alpha.assigned = True
    return pareto_set


def deep_copy_data(map_v):
    """The map that we want to perturb is the input. We return copies of
       stateG,
    dist_list, and cdi_data so that we don't unintentionally change data
       for map_v."""
    #map_u = deepcopy(map_v)
    stateG = deepcopy(map_v.graph)
    dist_list = deepcopy(map_v.dist_list)
    cdi_data = deepcopy(map_v.cdi_data)
    new_alpha = deepcopy(map_v.alpha)
    return stateG, dist_list, cdi_data, new_alpha


def record_best_metrics(ov, count, pareto_set):
    """Finds the best metric in the archive at a given iteration"""
    best_pop = np.inf
    best_comp = np.inf
    best_eg = np.inf
    best_cdi = np.inf
    best_egu = np.inf
    best_mm = np.inf

    for plan in pareto_set:
        if plan.pop_dev < best_pop: best_pop = plan.pop_dev
        if plan.compactness < best_comp: best_comp = plan.compactness
        if plan.eg < best_eg: best_eg = plan.eg
        if plan.cdi_num < best_cdi: best_cdi = plan.cdi_num
        if plan.excess_GU_num < best_egu: best_egu = plan.excess_GU_num
        if plan.mm < best_mm: best_mm = plan.mm
    ov.best_pop[count.recomcount] = best_pop
    ov.best_comp[count.recomcount] = best_comp
    ov.best_eg[count.recomcount] = best_eg
    ov.best_cdi[count.recomcount] = best_cdi
    ov.best_excess_GU[count.recomcount] = best_egu
    ov.best_mm[count.recomcount] = best_mm
    return


def plot_plan(G, idx):
    """Plots one plan with county lines"""
    high_contrast_colors = ['blue', 'green', 'yellow', 'magenta', 'cyan',
       'orange', 'purple']
    cmap = mcolors.ListedColormap(high_contrast_colors)
    with open('county_boundaries.pkl', 'rb') as fp: countyG =
       pickle.load(fp)
    planx = 'plan' + f'{idx}'
    fig, ax = plt.subplots(figsize=(10,10))
    G.data.plot(column=planx, ax=ax, legend=True, cmap=cmap)
    countyG.data.boundary.plot(ax=ax, color="black")
    fig.show()


def plot_plans(G, min_idx=0, max_idx=None):
    """Plots all maps between min_idx and max_idx"""
    high_contrast_colors = ['blue', 'green', 'yellow', 'magenta', 'cyan',
        'orange', 'purple']
    cmap = mcolors.ListedColormap(high_contrast_colors)
    with open('county_boundaries.pkl', 'rb') as fp: countyG =
       pickle.load(fp)
    if max_idx == None:
        i=min_idx
        planx = 'plan' + f'{i}'
        while planx in G.data:
            fig, ax = plt.subplots(figsize=(10,10))
            G.data.plot(column=planx, ax=ax, legend=True, cmap=cmap)
            countyG.data.boundary.plot(ax=ax, color="black")
            fig.show()
            i += 1
            planx = 'plan' + f'{i}'
    else:
        for i in range(min_idx, max_idx):
            planx = 'plan' + f'{i}'
            fig, ax = plt.subplots(figsize=(10,10))
            G.data.plot(column=planx, ax=ax, legend=True)
            countyG.data.boundary.plot(ax=ax,color="black")
            fig.show()

def main_loop(args):
    procid = args[0]
    count = args[1]
    ip = args[2]
    ov = args[3]
    now = args[4]
    alpha = args[5]
    map0 = args[6]
    temp = args[7]
    tol = args[8]
    stateG = args[9]
    dist_list = args[10]
    cdi_data = args[11]
    pareto_set = args[12]
    rand_seed = args[13]
    rng = args[14]

    rng.seed(rand_seed)

    District.num_dists = ip.distcount
    District.ideal_pop = round(sum(District.pop_list(dist_list)) /
       District.num_dists)
    print(f"We are in subprocess {procid}")

    #Starting the main line of the Simulated Annealing Algorithm
    while count.recomcount < ip.total_iterations:
        acceptable_mets = False
        count.unacceptable_met_counter = 0 #Resets the
           unacceptable_met_counter
        while acceptable_mets == False: #Will be true if we produce metrics
           that are acceptable
            if temp == ip.temp: alpha_val = map0.alpha
            map_v = Map_class(stateG, alpha_val, dist_list, cdi_data)




            # RECOM STEP
            count.failed_recom_counter = 0  #Resets the failed_recom_counter
            recom_success_flag = False
            while recom_success_flag == False:
                while True: #This list selects a District Neighbor pair
                   where one district is above the ideal pop and the other
                   is below
                    dist1 = rng.randint(0, District.num_dists - 1)
                    dist2 = rng.choice(dist_list[dist1].Dist_nbrs(stateG))
                    d1pop = dist_list[dist1].Population
                    d2pop = dist_list[dist2].Population
                    if np.sign(d1pop - District.ideal_pop) != np.sign(d2pop
                       - District.ideal_pop): break #districts are on
                       opposite sides of ideal_pop
                dist1, dist2, recom_success_flag = recom(rng, map_v, tol,
                   count, dist1=dist1, dist2=dist2, procid=procid)  #Does
                   recombination algorithm and returns "True" if recom
                   succeeded
                if recom_success_flag == False: count.failed_recom_counter
                   += 1
                if count.failed_recom_counter >= ip.max_failed_recoms:
                    print(f"PI{procid}. We failed in
                       {count.failed_recom_counter} consecutive recom
                       attempts. Skipping this recom step.")
                    break
            count.recomcount += 1
            if recom_success_flag == True:
                ov.fill_obj_vals(ip, dist_list, cdi_data, "recom",
                   count.currentit)
            else:
                ov.fill_obj_vals(ip, dist_list, cdi_data, "failed_recom",
                   count.currentit)

            # FLIP STEPS
            for _ in range(ip.num_flips):
                count.failed_flip_counter = 0  #Resets the
                   failed_flip_counter
                flip_success_flag = False
                while flip_success_flag == False and
                   count.failed_flip_counter < ip.max_failed_flips:
                    flip_success_flag, sa_prob = flip(rng, map_v, temp, ip,
                       count, dist1, dist2, procid)  #Does the Flip
                       algorithm and returns "True" if flip succeeded
                    if flip_success_flag == False:
                        count.failed_flip_counter += 1
                count.flipcount += 1
                if flip_success_flag == True:
                    ov.fill_obj_vals(ip, dist_list, cdi_data, "flip",
                       count.currentit, sa_prob)
                else:
                    #print(f"PI{procid}. R{count.recomcount}. Flip
                       algorithm failed after {count.failed_flip_counter}
                       attempts. Skipping this flip.")
                    ov.fill_obj_vals(ip, dist_list, cdi_data,
                       "failed_flip", count.currentit, sa_prob)



            ip.pop_max(stateG) #Sets the maximum for population deviation
            if all([map_v.metrics[i] <= ip.met_maxes[i] for i in
                range(len(ip.met_maxes)) if ip.metrics_to_use[i] == 1]):
                acceptable_mets = True
            ##May want to add an input possibility instead of 5
            elif count.unacceptable_met_counter < 5: #if some metric is
                unacceptable. Will recreate the map
                acceptable_mets = False
                count.unacceptable_met_counter += 1
                print("Unacceptable metrics found. Redoing the recom and
                   flips.")
                count.recomcount -= 1
                count.flipcount -= ip.num_flips
            else:
                acceptable_mets = False
                print(f"Unacceptable maps were produced {5} times in a row.
                   Recording this as a failed map.")
                break #breaks from 'while acceptable_mets == False' loop
                #ov.fill_obj_vals(ip, dist_list, cdi_data,
                   "unacceptable_metrics", count.currentit)
       if acceptable_mets == True:
            code, map_p = map_v.compare_objs(rng, pareto_set, ip)
       else:
            code = -3
            map_p = None

       if code == 1:  #If perturbed map is dominant
            pareto_set = replace_map(rng, pareto_set, map_p, map_v)
                #Replaces map_p with map_v in PS
            # if isinstance(map_p, list):
            #     count.alphacount -= len(map_p) - 1
            stateG, dist_list, cdi_data, alpha_val = deep_copy_data(map_v)
            print(f"PI{procid}. map_v is dominant over a map in the Pareto
                set. Adding map_v to the Pareto set, and removing
                {len(map_p)} map(s).")
            ov.map_conclusion_vals[count.recomcount] = f"PI{procid}.
                Dominant. Removed {len(map_p)} map(s)"

       elif code == 0 and len(pareto_set) < ip.num_maps:  #If the
            perturbed map is neither dominated nor dominant over any PS
            maps and there are fewer than 10 maps in the PS
            try:
                unassigned_alphas = [x for x in alpha if x.assigned ==
                   False]
                map_v.alpha = rng.choice(unassigned_alphas)
                #map_v.alpha = alpha[count.alphacount]
            except IndexError:  #In case we didn't build enough alpha
                values in the beginning
                add_alpha_row(alpha)
                # map_v.alpha = alpha[count.alphacount]
            # count.alphacount += 1
            pareto_set.append(map_v)
            stateG, dist_list, cdi_data, alpha_val = deep_copy_data(map_v)
            print(f"PI{procid}. map_v is middling. Adding map_v to the
                Pareto set since there aren't yet {ip.num_maps} maps in the
                PS.")
            ov.map_conclusion_vals[count.recomcount] = "Middling. Added to
                archive (not full)"

       elif (code == 0 and len(pareto_set) >= ip.num_maps) or code == -2:
            #If the perturbed map is neither dominated nor dominant over
            any PS maps and there are at least 10 maps in the PS
            if code == -2:
                warnings.warn("All maps in the Pareto set match the
                   perturbed map. This generally should not happen.")

            map_p = rng.choice(pareto_set)  #A random map from the Pareto
                set
            add_to_PS, sa_prob = sa_prob_calc(rng, map_v, map_p, temp, ip,
                count)
            if add_to_PS == True:
                pareto_set = replace_map(rng, pareto_set, map_p, map_v)
                print(f"PI{procid}. map_v is middling. By the SA
                   probability, we add this to the PS.")
                ov.map_conclusion_vals[count.recomcount] = "Middling. Added
                   to archive (SA Prob)"
            else:
                print(f"PI{procid}. map_v is middling. By the SA
                   probability, we DO NOT add this to the PS.")
                ov.map_conclusion_vals[count.recomcount] = "Middling. Not
                   added to archive (SA Prob)"

            accept_perturbation, sa_prob = sa_prob_calc(rng, map_v, map_p,
                temp, ip, count)
            if accept_perturbation == True:
                stateG, dist_list, cdi_data, alpha_val =
                   deep_copy_data(map_v)
                print(f"PI{procid}. We will continue to make perturbations
                   to map_v.")
            else:
                stateG, dist_list, cdi_data, alpha_val =
                   deep_copy_data(rng.choice(pareto_set))
                print(f"PI{procid}. Discarding map_v and reselecting a map
                   from the Pareto set.")

       elif code == -1:  #If the perturbed map is dominated by some map in
            the pareto set
```

```
            accept_perturbation, sa_prob = sa_prob_calc(rng, map_v, map_p,
                temp, ip, count)
            print(f"PI{procid}. map_v is dominated by a map in the Pareto
                set. We will not add it to the PS.")
            ov.map_conclusion_vals[count.recomcount] = "Dominated. Not
                added to archive"
            if accept_perturbation == True:
                stateG, dist_list, cdi_data, alpha_val =
                    deep_copy_data(map_v)
                print(f"PI{procid}. We will continue to make perturbations
                    to map_v.")
            else:
                stateG, dist_list, cdi_data, alpha_val =
                    deep_copy_data(rng.choice(pareto_set))
                print(f"PI{procid}. Discarding map_v and reselecting a map
                    from the Pareto set.")
        elif code == -3:  #If the map had a metric that was unacceptable
            ov.map_conclusion_vals[count.recomcount] = "Unacceptable
                Metrics. Not added to archive"

        else:
            raise RuntimeError(f"Unexpected code returned. code = {code}")

        record_best_metrics(ov, count, pareto_set)
        ov.hvol[count.recomcount] = hypervolume(pareto_set, ip)

        #print("\n")

        #Reduces temperature and tolerance if we've done the proper number
            of iterations at this temperature
        count.its_at_temp += 1
        if count.its_at_temp >= ip.num_its_per_temp:
            count.its_at_temp = 0
            temp = temp * ip.coolingrate
            tol = tol * ip.tol_coolingrate
    return pareto_set, ov


def main(*args):
    """Runs the primary instance of the algorithm."""
    rng = random.Random(args[0])
    print(f"random seed is {args[0]}.")


    #Get user input
    sig = signature(input_vals.__init__)

    if len(sys.argv) > 2:
        warnings.warn("Too many arguments were provided. Only the first
            will be considered")
    if len(sys.argv) > 1:  #If any argument was provided on the command line
        input_file_name = sys.argv[1]
        with open(input_file_name) as f:
            input_data_from_file = [line.strip() for line in f.readlines()]
        print(input_data_from_file)
        if len(input_data_from_file) != len(sig.parameters)-1: #-1 because
            of "self"
            raise ValueError(f"Incorrect number of arguments given in the
                input text document. Need {len(sig.parameters)-1}. Have
                {len(input_data_from_file)}")
        ip = input_vals(*input_data_from_file)
        print(ip.__dict__.values())





    elif len(args) == len(sig.parameters):
        print("Using args")
        ip = input_vals(args)  #Second, tries to take input from explicit
            input into main()
    else:
        print("Using default variable choices")
        #Inserting dummy values to be overwritten in next line
        None_list = [None] * (len(sig.parameters)-1)
        ip = input_vals(*None_list)
        ip.default_user_input()  #Finally, manually assigns input values if
            they aren't provided

    #Marking the start time of the run.
    now = datetime.datetime.now()
    print("Starting date and time : {}".format(now.strftime("%m-%d-%y
       %H:%M:%S")))
    timetxt = now.strftime("_%m%d%y_%H%M%S_%f")

    #This builds alpha, which is the normalized unit vector that details
       how much we care about any given metric.
    metric_count = sum(ip.metrics_to_use)
    alpha = build_alpha(rng, metric_count, ip)

    tol = ip.tol  #tol will be modified later
    temp = ip.temp  #temp will be modified later

    #Creates an initial map
    while True:
        if ip.preselected_start_map == False: #If we want to use random
            starting map do these steps, we initialize a random map
            stateG = initialize_map3(rng, ip)
        else:
            raise ValueError("preselected_start_map must be False for now")

        #Initializes pop_dev maximum
        ip.pop_max(stateG)

        #Creates an instance of District for each District
        dist_list = [None] * (ip.distcount)
        for i in range(ip.distcount):
            dist_list[i] = District(i)  #Initializes District variables for
                each district
        populate_dist_list(stateG, dist_list)
        # for idx, use_met in enumerate(ip.metrics_to_use):
        #     if use_met == 1:
        # viable_objs = [dist_list.metrics]


        #Populates County-District-Intersection (CDI) values
        cdi_data = CDI(stateG)
        print(f"CDI_Count = {cdi_data.cdi_count}")
        print(f"Total number of precincts (calculated by
            np.sum(cdi_data.cdi_mat)) = {np.sum(cdi_data.cdi_mat)}")

        #Creates vectors of zeros that will hold values for population
            deviation, average compactness, etc.
        ov = objective_vals(ip)

        #Populates the zeroth entry for all vectors
        ov.fill_obj_vals(ip, dist_list, cdi_data, "initialization", 0)

        #Initializing the main line of the Simulated Annealing Algorithm
        count = counters()  #Keeps track of the various counters needed
        pareto_set = []  #The list that will contain all high-quality maps
            in the Pareto Set

        #Populates zeroth entry for the pareto set
        map0 = Map_class(stateG, alpha[0], dist_list, cdi_data)
        under_ub = map0.check_objs(ip)
        if set(under_ub) == {True}: break #Breaks if all metrics are
            beneath upper bounds


    count.alphacount = 1  #We used the first alpha value in the previous
        line
    pareto_set.append(map0)
    ov.map_conclusion_vals[0] = "Initialization"
    record_best_metrics(ov, count, pareto_set)
    stateG = deepcopy(stateG)  #Creates a new instance of stateG so that
        future changes won't affect map0
    dist_list = deepcopy(dist_list)
    cdi_data = deepcopy(cdi_data)

    #################################### PROCESS POOL EXECUTOR
        ################################################
    procid = []
    rand_seeds = []
    length_to_r_list = rng.sample(range(800, 1000), ip.num_threads)
    rand_list = rng.sample(range(99999999), ip.num_threads*1001)
    l = 0
    for i in range(ip.num_threads):
        l += length_to_r_list[i] #How far in the random list we go to
            retrieve a random number
        procid.append(i) #The process ID for each process
        rand_seeds.append(rand_list[l]) #Gives each process its own random
            seed
    args_main_loop = []
```

```
pareto_set_list = []
ov_list = []
for i in range(ip.num_threads):
    #Each entry of args_main_loop is a list containing the values
       needed for the main loop
    args_main_loop.append([procid[i], count, ip, ov, now, alpha, map0,
       temp, tol, stateG, dist_list, cdi_data, pareto_set,
       rand_seeds[i], rng])
if ip.parallel == True:
    print("Starting Parallel Processes")
    with concurrent.futures.ProcessPoolExecutor() as executor:
        results = [executor.submit(main_loop, args_main_loop[i]) for i
           in range(ip.num_threads)]
    for i in results:
        ps = i.result()[0]
        ov_i = i.result()[1]
        pareto_set_list.append(ps) #Compiles all pareto sets from each
           parallel run into one list
        ov_list.append(ov_i) #Compiles all objective value lists into
           one list
    print("Parallel processes finished")
else:
    results, ov = main_loop(args_main_loop[0])
    pareto_set_list = results
    print("Main loop finished")


########################################################################

ending_time = datetime.datetime.now()
elapsed_time = ending_time - now
input_data = [list(ip.__dict__.values())]
input_columns = list(ip.__dict__.keys())
input_data[0].append(rand_seeds)
input_columns.append("rand_seeds")
input_data[0].append(elapsed_time.seconds)
input_columns.append("time expended")
df0 = pd.DataFrame(input_data, columns=input_columns)

print("Starting data:")
starting_data = [[map0.pop_dev, map0.compactness, map0.eg,
   map0.cdi_num, map0.excess_GU_num, map0.mm]]
df1 = pd.DataFrame(starting_data, columns = ["Population Deviation",
   "Compactness", "Efficiency Gap", "CDI", "excess GU", "Median Mean"])
print(df1)


data_table = []
if not isinstance(pareto_set_list, list): pareto_set_list =
   [pareto_set_list] #For non-parallel runs
if not isinstance(pareto_set_list[0], list): pareto_set_list =
   [pareto_set_list] #For non-parallel runs
map_list = [item for sublist in pareto_set_list for item in sublist]
for m in map_list: #'m' for 'map'
```

```
    wv = list(np.around(np.array(m.alpha.wv),3)) #Rounds the weight
        vector to 3 decimal places
    data_table.append([m.pop_dev, m.compactness, m.eg, m.cdi_num,
        m.excess_GU_num, m.mm, wv])

data_table_keep = [None] * len(data_table)
for m1_idx, map1 in enumerate(data_table):
    for m2_idx, map2 in enumerate(data_table[:m1_idx] +
        data_table[m1_idx+1:]): #Cycles through data_table *except* for
        map1
        m1_m2_comparison = []
        m1_m2_comparison.append(np.sign(map1[0] - map2[0])) #pop_dev
        m1_m2_comparison.append(np.sign(map1[1] - map2[1])) #compactness
        m1_m2_comparison.append(np.sign(map1[2] - map2[2])) #eg
        m1_m2_comparison.append(np.sign(map1[3] - map2[3])) #cdi_num
        m1_m2_comparison.append(np.sign(map1[4] - map2[4]))
            #excess_GU_num
        m1_m2_comparison.append(np.sign(map1[5] - map2[5])) #mm
        for idx, metric in enumerate(ip.metrics_to_use):
            if metric == 0:
                m1_m2_comparison[idx] = None
        m1_m2_comparison = remove_nones(m1_m2_comparison)
        if set(m1_m2_comparison) == {1}: #All metrics in map1 are worse
            than map2
            data_table_keep[m1_idx] = 0 #Indicates that we will NOT
                keep this map; it is dominated
            break
        elif set(m1_m2_comparison) == {0} and m2_idx >= m1_idx: #All
            metrics in map1 are the same as map2
            data_table_keep[m1_idx] = 1 #We keep this (duplicate) map.
                We keep the first duplicate
            continue
        elif set(m1_m2_comparison) == {0} and m2_idx < m1_idx: #All
            metrics in map1 are the same as map2
            data_table_keep[m1_idx] = 0
            break
        if m2_idx == len(data_table[:m1_idx] + data_table[m1_idx+1:]) -
            1: #If we reached the final map in the m2 loop
            data_table_keep[m1_idx] = 1 #Keep this entry in the PS

for idx, i in enumerate(data_table_keep):
    if i == 0:
        data_table[idx] = None
        map_list[idx] = None
data_table = remove_nones(data_table)
map_list = remove_nones(map_list)

df2 = pd.DataFrame(data_table, columns = ["Population Deviation",
    "Compactness", "Efficiency Gap", "CDI", "excess GU", "Median Mean",
    "Weight Vector"])
print(df2)


######################################################################
#Printing to Excel
```

```
met_string = ''
for idx, met in enumerate(ip.metrics_to_use):
    if met==1: met_string += 'O'
    else: met_string += 'X'

excel_doc_name = f"MOSA2_" + met_string + timetxt + ".xlsx"
col_1_header = "GU Number"
ps_col_headers = []
for i in range(len(map_list)):
    ps_col_headers.append(f"Plan {i}")
node_list = list(stateG.nodes)
node_list = [x+1 for x in node_list]  #+1 so nodes start at 1, not 0.
excel_dict = {}
excel_dict[col_1_header] = node_list
for i in range(len(map_list)):
    excel_dict[ps_col_headers[i]] =
        list(dict(map_list[i].graph.nodes("District Number")).values())
        #Gets the district assignments
df3 = pd.DataFrame(excel_dict)

#Hypervolume
if ip.parallel:
    df4_dict = {}
    for i in range(len(ov_list)):
        df4_dict[f'hypervolume thread {i}'] = [ov_list[i].hvol[it] for
            it in range(ip.num_recoms)]
    df4 = pd.DataFrame(df4_dict)
    final_hv_val = hypervolume(map_list, ip)
    final_hv_row = pd.DataFrame([[final_hv_val] * df4.shape[1]],
        columns=df4.columns)
    df4 = pd.concat([df4, final_hv_row], ignore_index=True)
else: #not parallel
    df4_dict = {} #Contains generational measures
    df4_dict['hypervolume'] = [ov.hvol[it] for it in
        range(ip.num_recoms)]
    df4 = pd.DataFrame(df4_dict)

excel_writer = pd.ExcelWriter(excel_doc_name)

df0.to_excel(excel_writer, sheet_name='Input Data')
df1.to_excel(excel_writer, sheet_name='Starting Data')
df2.to_excel(excel_writer, sheet_name='Metrics')
df3.to_excel(excel_writer, sheet_name='District Assignments')
df4.to_excel(excel_writer, sheet_name='Hypervolume')

excel_writer.save()

######################################################################
# Plotting results
if ip.parallel:
    pf = map_list #pf = pareto front
else:
    pf = pareto_set
    #Plot Plans
    psda = [None]*len(pf)
    for i in range(len(pf)):
        psda[i] = df3[["GU Number", f"Plan {i}"]] #psda=pareto set dist
            assignment
        pf[i].graph.data = pf[i].graph.data.merge(right=psda[i],
            left_on='OBJECTID', right_on='GU Number', validate='1:1') #adds
            this data to pareto_set graph df
        pf[i].graph.data = pf[i].graph.data.drop('GU Number', axis=1)
            #deletes 'GU Number' column
        pf[i].graph.data = pf[i].graph.data.rename(columns={f'Plan {i}':
            f'plan{i}'}) #Shortens the plan name column
        plot_plan(pf[i].graph, i)



    ######################################################################


    print("Ending date and time : {}".format(ending_time.strftime("%m-%d-%y
       %H:%M:%S")))
    print(f"Elapsed time = {elapsed_time}")
    print("Finished!")

if __name__ == "__main__":
    rseed = secrets.randbelow(sys.maxsize)
    #rseed = 7066382009700737341
    main(rseed)
```

Appendix D NSGA-II Codes

```
    from gerrychain import Graph
import geopandas as gpd
import networkx as nx
import time
import datetime
import math
import random
import numpy as np
import pickle
import pandas as pd
import matplotlib.pyplot as plt
import sys
import plotly.express as px
import secrets
from pymoo.indicators.hv import HV
import copy
import matplotlib.colors as mcolors
#from pyMultiobjective.utils import indicators
#from deap.tools import hypervolume as hv
#from deap.benchmarks.tools import hypervolume
#import hv



class input_c: #short for 'input class'
    def __init__(self, json_file, ps, mp, nd, ng, met, cf, lmn, rnd, rad):
        self.json_file = json_file #Input file for the graph
        self.pop_size = ps #Population size
        self.mutation_probability = mp #Mutaiton probability
        self.num_districts = nd #Number of districts
        self.num_generations = ng #Number of generations
        self.metrics = met #Metrics dictionary. Contains either True or
           False for each metric
        self.county_file = cf #File countaining counties
        self.long_met_names = lmn #Long metric names
        self.randomness = rnd #Randomness level
        self.radius = rad #Radius for contiguity checks. Higher values are
           more thorough.

        # Max cluster num. Entries will be (gen, pop_id): max_cl_num
        self.max_cl_num = {}

    def __repr__(self):
        met_string = ''
        for met in self.metrics:
            if self.metrics[met]==True: met_string += 'O'
            else: met_string += 'X'
        return f"<INPUT: pop_size = {self.pop_size}, num_gens =
           {self.num_generations}, mets = [{met_string}]>"

    @classmethod
    def Load_from_file(cls, file_name):
        """Reads the input from a file"""
        with open(file_name) as f:
            lines = f.readlines()
        json_file = lines[0].strip()
        pop_size = int(lines[1])
        mut_prob = float(lines[2])
        num_districts = int(lines[3])
        num_generations = int(lines[4])
        metrics = {"dpop": eval(lines[5].strip().lower().capitalize()),
                   "comp": eval(lines[6].strip().lower().capitalize()),
                   "eg": eval(lines[7].strip().lower().capitalize()),
                   "mm": eval(lines[8].strip().lower().capitalize()),
                   "cs": eval(lines[9].strip().lower().capitalize()),
                   "egu": eval(lines[10].strip().lower().capitalize())
                   }
        county_file = lines[11][0:-1]
        met_long_names = {"dpop": lines[12].strip(),
                          "comp": lines[13].strip(),
                          "eg": lines[14].strip(),
                          "mm": lines[15].strip(),
                          "cs": lines[16].strip(),
                          "egu": lines[17].strip(),
                         }
        randomness = int(lines[18])
        radius = float(lines[19])
        return cls(
            json_file=json_file,
            ps=pop_size,
            mp=mut_prob,
            nd=num_districts,
            ng=num_generations,
            met=metrics,
            cf=county_file,
            lmn=met_long_names,
            rnd=randomness,
            rad=radius
        )


def total_pop(G, ip):
    """Calculates the total population of the graph"""
    try:
        return ip.total_pop
    except AttributeError:
        ip.total_pop = sum(dict(G.nodes("POPULATION")).values())
        return ip.total_pop


def target_pop(G, ip):
    """Calculates the target population for each district"""
    try:
        return ip.ideal_pop
    except AttributeError:  # In the event that ideal_pop has not yet been
       calculated
        ip.total_pop = sum(dict(G.nodes("POPULATION")).values())
        ip.ideal_pop = ip.total_pop/ip.num_districts
        return ip.ideal_pop

def make_nbr_graphs(G, radius=3):
    """Makes neighbor graphs for every node at a distance of 'radius' away.
    G: Base graph.
    radius : Include all neighbors of distance<=radius from n. Default 3"""
    if radius == float('inf'): radius = nx.diameter(G) #if radius is
       infinity, redefine to its diameter
    nbr_graphs = {}
    for n in G:
        nbr_graphs[n] = nx.ego_graph(G, n, radius=radius) #All
            neighborhoods of radius 3 for each node
    return nbr_graphs

def initialize_graph(ip):
    """Builds a graph based on a json file supplied by the user"""
    try:
        with open('stateG.pkl', 'rb') as fp: G = pickle.load(fp)
    except FileNotFoundError:
        cols_to_add = ['OBJECTID', 'ID', 'VTD', 'COUNTY', 'STATE', 'NAME',
            'POPULATION',
                    'CountyName', 'Reg_Voters', 'PresBlue', 'PresRed',
                       'PresOther',
                    'SenBlue', 'SenRed', 'SenOther', 'Cong2011', 'Sen2011',
                       'House2011',
                    'Cong2021', 'Sen2021', 'House2021', 'geometry']
        try:
            G = Graph.from_file(
                ip.json_file,
                adjacency="rook",
                cols_to_add=cols_to_add,
                reproject="True",
                ignore_errors="True"
            )
        except KeyError:
            G = Graph.from_file(
                ip.json_file,
                adjacency="rook",
                cols_to_add=None,  # Will add all columns
                reproject="True",
                ignore_errors="True"
            )
        #Initialize the layer
        G.layer = 0

        #Initialize front[0] sizes
        G.front0_sizes = []

        #Save stateG if it hasn't been saved before
        with open('stateG.pkl', 'wb') as fp: pickle.dump(G, fp)

    #Initialize the metric keepers
    if ip.metrics['dpop'] == True:
        G.pop_dev = np.zeros((ip.num_generations+1, 2*ip.pop_size))
    if ip.metrics['comp'] == True:
        G.comp = np.zeros((ip.num_generations+1, 2*ip.pop_size))
    if ip.metrics['eg'] == True:
        G.eg = np.zeros((ip.num_generations+1, 2*ip.pop_size))
    if ip.metrics['mm'] == True:
        G.mm = np.zeros((ip.num_generations+1, 2*ip.pop_size))
    if ip.metrics['cs'] == True:
        G.cs = np.zeros((ip.num_generations+1, 2*ip.pop_size))
    if ip.metrics['egu'] == True:
        G.egu = np.zeros((ip.num_generations+1, 2*ip.pop_size))
    G.metrics = np.zeros((ip.num_generations+1, 2*ip.pop_size,
       sum(ip.metrics.values())))
    ip.num_GUs = G.number_of_nodes()
    county_list = set()
    for n in G.nodes:
        G.nodes[n]['gen'] = np.full(
            shape=(ip.num_generations+1, ip.pop_size*2, 2), fill_value=(-1,
              -1))
        # 2 represents that each entry of G.nodes[n]['gen'][g][c] will
           return a (clustering, districting) tuple
        county_list.add(G.nodes[n]['COUNTY'])
    county_list = sorted(county_list)
    # Long county name to short county name. e.g. longc2shortc['45003'] = 1
    longc2shortc = {}
    for idx, c in enumerate(county_list):
        longc2shortc[c] = idx
    for n in G.nodes:
        G.nodes[n]['county_num'] = longc2shortc[G.nodes[n]['COUNTY']]
    return G

def create_attr_graph(G, ip, attr=None, gen_id_cldi=None):
    """Creates a graph that merges all nodes of a certain attribute (attr)
       together
    'G' is NOT amended in this function
    gen_id_cldi will be entered only if 'attr' is not entered.
    gen_id_cldi represents the triplet (generation, clustering/districting
       pop_id, clustering/districting)
    'generation' sets the current generation [0,1000]
    'pop_id' sets the identification number for the clustering/districting
       pair [0,49]
    'cldi' sets whether we are grouping a clustering or a districting [0:
       clustering, 1: districting] """
    if attr == None and gen_id_cldi == None:
        raise ValueError("Either 'attr' or 'gen_id_cldi' must be defined")
    if attr != None and gen_id_cldi != None:
        raise ValueError("Either 'attr' or 'gen_id_cldi' must be defined,
           but not both")

    gen = None
    pop_id = None
    cldi = None
    if gen_id_cldi != None:
        gen = gen_id_cldi[0]
        pop_id = gen_id_cldi[1]
        cldi = gen_id_cldi[2]
```

```
cg = nx.Graph() #cg = contracted graph
cg.layer = G.layer + 1
node2supernode = {}
node_attr_list = [
    'area',
    'POPULATION',
    'Reg_Voters',
    'PresBlue',
    'PresRed',
    'PresOther',
    'SenBlue',
    'SenRed',
    'SenOther',
    ]
# Removes all edges from graph G where one end has different attribute
   (attr) values than the other end
#boundary_edges = []
for e in G.edges():
    n0 = e[0]
    n1 = e[1]

    if attr != None: # we are separating by attribute
        sn0 = G.nodes[n0][attr]
        sn1 = G.nodes[n1][attr]
    elif gen_id_cldi != None: #we are separating by either districting
       or clustering
        sn0 = G.nodes[n0]['gen'][gen][pop_id][cldi]
        sn1 = G.nodes[n1]['gen'][gen][pop_id][cldi]
    else: raise ValueError("Either attr or gen_id_cldi should be
       defined")
    node2supernode[n0] = sn0
    node2supernode[n1] = sn1

    if sn0 != sn1: #i.e, this edge represents a boundary for the
       attribute
        #boundary_edges += [e]

        # try: supernode2node[sn0].append(n0)
        # except KeyError: supernode2node[sn0] = [n0]
        # try: supernode2node[sn1].append(n1)
        # except KeyError: supernode2node[sn1] = [n1]
        cg.add_node(sn0)
        cg.add_node(sn1)
        cg.add_edge(sn0, sn1)

        #Adds shared perimeter to total perimeter attribute for nodes
        try: cg.nodes[sn0]['total_perim'] += G.edges[(n0,
           n1)]['shared_perim']
        except KeyError: cg.nodes[sn0]['total_perim'] = G.edges[(n0,
           n1)]['shared_perim']
        try: cg.nodes[sn1]['total_perim'] += G.edges[(n0,
           n1)]['shared_perim']
        except KeyError: cg.nodes[sn1]['total_perim'] = G.edges[(n0,
           n1)]['shared_perim']
        #Add edge attributes to cg *only* if it was a boundary edge
        try: cg[sn0][sn1]['shared_perim'] += G.edges[(n0,
            n1)]['shared_perim']
        except KeyError: cg.edges[(sn0, sn1)]['shared_perim'] =
            G.edges[(n0, n1)]['shared_perim']

        # Creates a list of all edges that created this supernode edge
        cg[sn0][sn1]['contributing_edges'] = []
        if cg.layer == 1: #If we're contracting the base graph
            cg[sn0][sn1]['contributing_edges'] += [e]
        else: #If we're contracting an already contracted graph
            cg[sn0][sn1]['contributing_edges'] = 'maybe add later?'

for sn in cg.nodes():
    cg.nodes[sn]['boundary_perim'] = 0
    cg.nodes[sn]['GUs'] = []
    if cg.layer == 2:
        cg.nodes[sn]['clgu'] = {}
        #cg.nodes[sn]['districting'] = sn

for n in G.nodes():
    sn = node2supernode[n]

    if cg.layer == 1: #Creating a clustering graph
        cg.nodes[sn]['GUs'].append(n)
        #if len(cg.nodes[sn]['GUs']) > 12 and attr != 'county_num':
        #    print("This is also a problem")
    elif cg.layer > 1: #Creating a districting graph
        if 'GUs' in cg.nodes[sn]: cg.nodes[sn]['GUs'] +=
            G.nodes[n]['GUs']
        else: cg.nodes[sn]['GUs'] = G.nodes[n]['GUs']

    #Initialize and populate the attributes
    for nattr in node_attr_list:
        if nattr in cg.nodes[sn]: cg.nodes[sn][nattr] +=
            G.nodes[n][nattr]
        else: cg.nodes[sn][nattr] = G.nodes[n][nattr]

    #For boundary_perim
    try:
        cg.nodes[sn]['boundary_perim'] += G.nodes[n]['boundary_perim']
        cg.nodes[sn]['total_perim'] += G.nodes[n]['boundary_perim']
    except KeyError: #If supernode isn't on a state boundary
        cg.nodes[sn]['boundary_perim'] += 0
        cg.nodes[sn]['total_perim'] += 0

    #For knowing the number of GUs in each county
    if cg.layer == 1:
        county_num = G.nodes[n]['county_num']
        try: cg.nodes[sn]['county_dict'][county_num] += 1
        except KeyError:
            cg.nodes[sn]['county_dict'] = {}
            cg.nodes[sn]['county_dict'][county_num] = 1
    # Names the county if the attribute is 'COUNTY' (done more times
        than necessary, but that's ok)
    if attr == 'COUNTY' or attr == 'county_num':
        cg.nodes[sn]['CountyName'] = G.nodes[n]['CountyName']

    #Establishing a cluster: GU dictionary if cg is a district graph
    if cg.layer == 2:
        cg.nodes[sn]['clgu'][n] = G.nodes[n]['GUs'] #n is a cluster.
            #sn is a district
        # if len(G.nodes[n]['GUs']) > 12:
        #     print("This is a big problem")

for sn in cg.nodes():
    if cg.layer == 1:  # indicating that we're grouping by clustering
        #cg.nodes[sn]['clustering'] = sn
        cg.nodes[sn]['districting'] = -1
        #cg.nodes[sn]['clgu'] = {}
        #cg.nodes[sn]['clgu'][sn] = cg.nodes[sn]['GUs'] # A dictionary
            with one entry. cluster: [GU_list]
        cg.nodes[sn]['counties'] = set([G.nodes[gu]['county_num'] for
            gu in cg.nodes[sn]["GUs"]])

# If we are creating the district plan, calculate the metrics
if cg.layer == 2:
    try: cg.total_pop = ip.total_pop
    except AttributeError:
        cg.total_pop = sum(dict(cg.nodes("POPULATION")).values())
        ip.total_pop = cg.total_pop
    try: cg.ideal_pop = ip.ideal_pop
    except AttributeError:
        cg.ideal_pop = cg.total_pop / ip.num_districts
        ip.ideal_pop = cg.ideal_pop
    # CDI_matrix[dist][county]
    CDI_matrix = np.zeros((ip.num_districts, ip.num_counties),
        dtype=np.int16)
    for d in cg.nodes: #'d' represents a district
        cg.nodes[d]['pp_compactness'] = 4 * math.pi *
            cg.nodes[d]['area'] / (cg.nodes[d]['total_perim'] ** 2)
        cg.nodes[d]['shifted_pp_compactness'] = 1 -
            cg.nodes[d]['pp_compactness']
        cg.nodes[d]['total_Pvotes'] = cg.nodes[d]['PresBlue'] +
            cg.nodes[d]['PresRed'] + cg.nodes[d]['PresOther']
        cg.nodes[d]['total_Svotes'] = cg.nodes[d]['SenBlue'] +
            cg.nodes[d]['SenRed'] + cg.nodes[d]['SenOther']
        cg.nodes[d]['blue_Pshare'] = cg.nodes[d]['PresBlue'] /
            cg.nodes[d]['total_Pvotes'] if
            cg.nodes[d]['total_Pvotes']>0 else 0
        cg.nodes[d]['red_Pshare'] = cg.nodes[d]['PresRed'] /
            cg.nodes[d]['total_Pvotes'] if
            cg.nodes[d]['total_Pvotes']>0 else 0
        cg.nodes[d]['other_Pshare'] = cg.nodes[d]['PresOther'] /
            cg.nodes[d]['total_Pvotes'] if
            cg.nodes[d]['total_Pvotes']>0 else 0
        cg.nodes[d]['blue_Sshare'] = cg.nodes[d]['SenBlue'] /
            cg.nodes[d]['total_Svotes'] if
            cg.nodes[d]['total_Svotes']>0 else 0
        cg.nodes[d]['red_Sshare'] = cg.nodes[d]['SenRed'] /
            cg.nodes[d]['total_Svotes'] if
            cg.nodes[d]['total_Svotes']>0 else 0
        cg.nodes[d]['other_Sshare'] = cg.nodes[d]['SenOther'] /
            cg.nodes[d]['total_Svotes'] if
            cg.nodes[d]['total_Svotes']>0 else 0
        cg.nodes[d]['win_threshold_P'] =
            math.ceil(cg.nodes[d]['total_Pvotes']/2 + 0.5)
        cg.nodes[d]['win_threshold_S'] =
            math.ceil(cg.nodes[d]['total_Svotes']/2 + 0.5)
        cg.nodes[d]['Rwasted_votes_P'] = cg.nodes[d]['PresRed'] -
            cg.nodes[d]['win_threshold_P'] if cg.nodes[d]['PresRed'] >
            cg.nodes[d]['PresBlue'] else cg.nodes[d]['PresRed']
        cg.nodes[d]['Bwasted_votes_P'] = cg.nodes[d]['PresBlue'] -
            cg.nodes[d]['win_threshold_P'] if cg.nodes[d]['PresRed'] <=
            cg.nodes[d]['PresBlue'] else cg.nodes[d]['PresBlue']
        for cl in cg.nodes[d]['clgu']:
            county_dict = G.nodes[cl]['county_dict']
            for c, v in county_dict.items(): #c = county, v = value
                (number of GUs in cluster-county intersection)
                CDI_matrix[d][c] += v
    cg.pop_dev_sum = round(sum(abs(p-cg.ideal_pop) for p in
        dict(cg.nodes("POPULATION")).values()))
    cg.pop_dev_max = round(max(abs(p-cg.ideal_pop) for p in
        dict(cg.nodes("POPULATION")).values()))
    cg.exc_county_splits = np.count_nonzero(CDI_matrix) -
        max(ip.num_counties, ip.num_districts)
    cg.exc_gus = sum(np.sum(CDI_matrix, axis=0) - np.max(CDI_matrix,
        axis=0))
    #cg.cbm = cg.exc_county_splits + ip.num_counties / ip.num_GUs *
        cg.exc_gus  # cbm = county-boundary-metric
    cg.cs = cg.exc_county_splits
    cg.egu = cg.exc_gus
    cg.pp_comp = round(sum([cg.nodes[d]['pp_compactness'] for d in
        cg.nodes]) / cg.number_of_nodes(), 3)
    cg.shifted_pp_comp = 1-cg.pp_comp
    bsP_list = [cg.nodes[d]['blue_Pshare'] for d in cg.nodes]  # bsP =
        blue share for president
    bsP_list = sorted(bsP_list)
    if ip.num_districts % 2 == 0:  # even
        bsP_median = (bsP_list[int(ip.num_districts/2)] +
            bsP_list[ip.num_districts/2-1])/2
    else:  # odd
        bsP_median = bsP_list[int(ip.num_districts/2-0.5)]
    bsP_mean = sum(bsP_list)/ip.num_districts
    cg.mm = abs(bsP_median - bsP_mean)  # median-mean
    bwv_P = sum([cg.nodes[d]['Bwasted_votes_P'] for d in cg.nodes])
    rwv_P = sum([cg.nodes[d]['Rwasted_votes_P'] for d in cg.nodes])
    cg.total_Pvotes = sum([cg.nodes[d]['total_Pvotes'] for d in
        cg.nodes])
    cg.eg = abs((bwv_P - rwv_P)/cg.total_Pvotes)  # absolute efficiency
        gap
    cg.all_metrics = [cg.pop_dev_sum, cg.shifted_pp_comp, cg.eg, cg.mm,
        cg.cs, cg.egu]
        cg.metric_keys = ['dpop', 'comp', 'eg', 'mm', 'cs', 'egu']
        cg.metrics = [m for idx, m in enumerate(cg.all_metrics) if
            ip.metrics[cg.metric_keys[idx]] == True]
    return cg

def initialize_clustering(rng, G, ip, pop_id, num_clusters=None):
    """Assigns each node a cluster number"""
    if num_clusters == None:
        num_clusters = math.ceil(G.number_of_nodes()/3.5)
    centers = rng.sample(list(G.nodes), num_clusters)
    unassigned_nodes = list(set(G.nodes()) - set(centers))
    rng.shuffle(unassigned_nodes)
    maxsize = {}  # The maximum size of each cluster
    currsize = {}  # The current size of each cluster; to be updated and
       not to exceed maxsize
    nodecluster = {} # A dictionary that will contain the list of nodes in
       each cluster. i.e., nodecluster[0] = [1,2,3] would mean that nodes
       1,2,3 are in cluster 0.
    cluster_nbrs = {} # A dictionary that contains all neighbors of nodes
       in the cluster
    # clusternum will identify the cluster number
    for clusternum, center in enumerate(centers):
        G.nodes[center]['gen'][0][pop_id][0] = clusternum
        #unassigned_nodes.remove(center)
        maxsize[clusternum] = rng.randint(1, 6)
        currsize[clusternum] = 1
        cluster_nbrs[clusternum] = set(G.neighbors(center))
        try:
            # appends to the list if it exists
            nodecluster[clusternum] += [center]
        except KeyError:
            # creates the list if it is brand new
            nodecluster[clusternum] = [center]
    rng.shuffle(centers)

    for center in centers:
        clusternum = G.nodes[center]['gen'][0][pop_id][0]
        while currsize[clusternum] < maxsize[clusternum]:
            currsize, maxsize, unassigned_nodes, nodecluster, _,
                cluster_nbrs, _ = grow_group( rng,
                G, ip, clusternum, 0, pop_id, currsize, maxsize,
                    unassigned_nodes, nodecluster, cluster_nbrs, None, None)
    while len(unassigned_nodes) > 0:  # While there are still unassigned
       nodes
        clusternum = max(currsize.keys()) + 1
        center = unassigned_nodes.pop()
        num_clusters += 1
        centers += [center]
        maxsize[clusternum] = rng.randint(1, 6)
        currsize[clusternum] = 1
        cluster_nbrs[clusternum] = set(G.neighbors(center))
        try:
            nodecluster[clusternum] += [center] # appends to the list if it
                exists
        except KeyError:
            nodecluster[clusternum] = [center] # creates the list if it is
                brand new
        G.nodes[center]['gen'][0][pop_id][0] = clusternum
        while currsize[clusternum] < maxsize[clusternum]:
            currsize, maxsize, unassigned_nodes, nodecluster, _,
                cluster_nbrs, _ = grow_group( rng,
                G, ip, clusternum, 0, pop_id, currsize, maxsize,
                    unassigned_nodes, nodecluster, cluster_nbrs, None, None)
    # Saves the maximum cluster number for the (gen, pop_id) pair
    ip.max_cl_num[(0, pop_id)] = max(currsize.keys())

def flip(G, n, dist, gen, pop_id, nbr_graphs, contig=float('inf')):
    """Performs a flip where we move GU 'n' to distrist 'dist'. This checks
        that contiguity will be maintained.
    G: Base graph
    n: Node to be flipped
    dist: district that n will be moved into
    gen: current generation
    pop_id: the population ID for the flip
    nbr_graphs: the k-layer neighbor graph dictionary for all nodes.
        nbr_graphs[n] = [list of nodes k away from n]
    contig: The level of contiguity to check. Can be float('inf') or any
        positive integer.
        If float('inf'), then we collect all GUs in current district and
            see if removing n creates discontiguity.
        If a positive integer 'k' is used, we only check GUs in the current
            district that are at most k away from n, and see if removing n
            creates discontiguity."""
    n_dist = G.nodes[n]['district'][gen][pop_id]
    if contig == float('inf'): #Check full contiguity
        nbrhd = nx.subgraph(G, [x for x in G.nodes() if
            G.nodes[x]['district'][gen][pop_id] == n_dist and x!=n])
    else: #Check k-layer contiguity
        nbrhd = nx.subgraph(nbr_graphs[n], [x for x in
            nbr_graphs[n].nodes() if G.nodes[x]['district'][gen][pop_id] ==
            n_dist and x!=n])
    if nx.is_connected(nbrhd):
        G.nodes[n]['district'][gen][pop_id] = dist
        return True #signifies flip was successful
    else: #flip would create discontiguity
        return False #signifies flip was unsuccessful

def grow_group(rng, G, ip, groupnum, gen, pop_id, currsize, maxsize,
    unassigned_nodes, nodegroup, group_nbrs, nbrlengths, viable_dists):
    """Grows the group (cluster/district) 'groupnum' by one GU (or cluster)
        unless it can't
    G: Graph
    ip: input
    groupnum: the group number to be expanded
    gen: The generation we're on
    pop_id: The pop_id of the group under consideration
    currsize: a dictionary containing the current size (in nodes) of each
        group
    maxsize: a dictionary containing the the maximum size (in nodes) of
        each group
```

```
unassigned_nodes: A list of nodes that have not yet been assigned a
    group
nodegroup: A dictionary detailing all the nodes in each cluster/district
group_nbrs: A dictionary detailing all the nbrs of nodes in the group
    (group_nbrs[0] = {set of nbr nodes})
nbrlengths: A dictionary detailing the boundary length of each
    district's neighbors. Only used for districts.
     ex: nbrlengths[0][646] would return the boundary length between
         cluster 646 and district 0.
viable_dists: Not used, but lists all the viable districts that could
    be expanded"""

if G.layer != 0 and G.layer != 1: raise ValueError("G's layer must be 0
    or 1")
if groupnum == -1: raise ValueError("groupnum should be nonnegative")

if nbrlengths == None:
    nbrs = list(group_nbrs[groupnum]) #Occurs when building *clusters*
else:
    nbrs = list(nbrlengths[groupnum].keys())

if G.layer==0: #For building clusters
    nbrs = nbrs
elif len(nbrs) > 0: #For building districts
    if ip.randomness == 0:
        nbrs = nbrs
    elif ip.randomness == 1:
        nbrs = weighted_shuffle(rng, nbrs,
            weights=list(nbrlengths[groupnum].values()))
    elif ip.randomness == 2:
        nbrs = nbrs

i=-1 #for cycling for nbrs indices
adj_max_size = True #flag relevant for district building
while True: # We break once a nbr is found or all nbrs are exhausted
    if ip.randomness<=1:
        if len(nbrs) == 0: break
        i += 1
        nbr = nbrs[i]
    elif ip.randomness==2: #If nbrs are chosen at random
        nbr_idx = rng.choice(range(len(nbrs)))
        nbr = nbrs[nbr_idx]

    if G.layer == 0:
        nbr_groupnum = G.nodes[nbr]['gen'][gen][pop_id][0] #Growing a
            cluster
        nbr_county = G.nodes[nbr]['county_num']
    elif G.layer == 1:  # Occurs if G was not the base graph with GUs
        nbr_groupnum = G.nodes[nbr]['districting'] #Growing a district
        nbr_county = G.nodes[nbr]['counties']

    if nbr_groupnum == -1:  # If the nbr has not been assigned a group
        already
        if G.layer == 0:
            r = rng.random() #for county purposes
```

```
                n = nodegroup[groupnum][0] #a node in the current cluster
                    (I think this will necessarily be the center)
                #If (county splits are important and counties don't match
                    and 1% probability hits) or (CS aren't important) or
                    (CS are important and counties match)
                if (ip.metrics['cs'] and
                    nbr_county!=G.nodes[n]['county_num'] and r<=0.01) or
                    ip.metrics['cs']==False or (ip.metrics['cs'] and
                    nbr_county== G.nodes[n]['county_num']):
                     G.nodes[nbr]['gen'][gen][pop_id][0] = groupnum
                elif i == len(nbrs)-1: break #breaks if we've searched all
                    neighbors
                else: continue #search for a new nbr
            elif G.layer == 1: # Occurs if G was not the base graph with GUs
                r = rng.random() #for county purposes
                n = nodegroup[groupnum][0] #a node in the current district
                    (I think this will necessarily be the center)
                #If (county splits are important and counties don't match
                    and 25% probability hits) or (CS aren't important) or
                    (CS are important and counties match)
                if (ip.metrics['cs'] and nbr_county!=G.nodes[n]['counties']
                    and r<=0.25) or ip.metrics['cs']==False or
                    (ip.metrics['cs'] and nbr_county==
                    G.nodes[n]['counties']):
                     G.nodes[nbr]['districting'] = groupnum
                elif i == len(nbrs)-1:
                     adj_max_size = False #we do NOT adjust the maxsize if
                         we break from this search.
                     break #breaks if we've searched all neighbors
                else: continue #search for a new nbr
            nodegroup[groupnum] += [nbr]
            unassigned_nodes.remove(nbr)
            currsize[groupnum] += 1
            nbrlengths = update_nbrlengths(G, ip, nbrlengths, nbr, groupnum)

            if nbrlengths != None and nbrlengths[groupnum] == {}:
                maxsize[groupnum] = currsize[groupnum]
            return currsize, maxsize, unassigned_nodes, nodegroup, nbr,
                group_nbrs, nbrlengths
        else: # If the nbr has been assigned
            if ip.randomness<=1:
                if i == len(nbrs)-1:
                    break #breaks from 'while True' loop
            else:
                nbrs[nbr_idx] = nbrs[-1]
                nbrs.pop()
                if len(nbrs) == 0: break
    # If we made it this far, it's because we checked all neighbors and
        can't append any to the current group
    if adj_max_size: maxsize[groupnum] = currsize[groupnum] # Adjusts the
        maxsize of this group so we don't iterate through its neighbors
        again
    return currsize, maxsize, unassigned_nodes, nodegroup, None,
        group_nbrs, nbrlengths
```

```
def weighted_shuffle(rng, items, weights, rvs=True,):
    """Given a list of items and weights for each one, we return a list of
       those shuffled items where higher weights are more probable"""
    order = sorted(range(len(items)), key=lambda i: rng.random() ** (1.0 /
       weights[i]), reverse=rvs)
    return [items[i] for i in order]

def update_nbrlengths(G, ip, nbrlengths, newnbr, dist):
    """Updates the nbrlengths dictionary after newnbr was added to
       district"""
    if nbrlengths == None: return None #Occurs when we are in the
       initialize_clustering function
    del nbrlengths[dist][newnbr] #Since newnbr is in dist now, dist doesn't
       neighbor newnbr
    for nn in G.neighbors(newnbr):
        if G.nodes[nn]['districting'] == -1:
            try:
                nbrlengths[dist][nn] += G.edges[(nn,
                    newnbr)]['shared_perim']
            except KeyError:
                nbrlengths[dist][nn] = G.edges[(nn, newnbr)]['shared_perim']
        elif G.nodes[nn]['districting'] != dist:
            nndist = G.nodes[nn]['districting']
            try: del nbrlengths[nndist][newnbr] #Occurs if nn is already
               assigned a district
            except KeyError: pass
    if ip.randomness <= 1:
        nbrlengths[dist] = dict(sorted(nbrlengths[dist].items(), key=lambda
           x:x[1], reverse=True))
    return nbrlengths

def clear_clustering(G, gen, pop_id, stateG=None):
    """For each node n, clears entered value for
       G.nodes[n]['gen'][gen][pop_id][cldi] and replaces it with -1
    G: cluster graph
    gen: current generation
    pop_id: the population id to be cleared
    stateG: the underlying GU graph"""
    if G.layer==1:  # If the input graph isn't the stateG graph
        for n in G.nodes:
            G.nodes[n]['districting'] = -1
    if stateG != None:
        for n in stateG.nodes:
            stateG.nodes[n]['gen'][gen][pop_id][1] = -1

def node_list_by_attr(G, attr_str, attr_val):
    """Returns a list of all nodes matching the given attribute."""
    try: return G.boundary_nodes
    except AttributeError:
        G.boundary_nodes = [n for n in G if G.nodes[n][attr_str] ==
           attr_val]
        return G.boundary_nodes

def node_list_by_cluster(G, gen, pop_id, cl_val):
    """Returns a list of all nodes matching the given cluster pop_id and
        generation"""
    return [n for n in G if G.nodes[n]['gen'][gen][pop_id][0] == cl_val]

def node_list_by_district(G, gen, pop_id, dist_val):
    """Returns a list of all nodes in the given district at the provided
        generation"""
    return [n for n in G if G.nodes[n]['gen'][gen][pop_id][1] == dist_val]

def create_cut_line(rng, G, sp, spl, n1=None, n2=None):
    """Finds the shortest path between node n1 and n2. The resultant cut
        line must split the graph into exactly 2 pieces.
    sp = shortest_paths
    spl = shortest path lengths
    returns the two halves of the graph and the cut_line after the cut line
        splits the graph"""
    if n1 == None and n2 != None:
        raise ValueError("If n2 is specified, n1 must be too")
    nodes_specified = False
    if n1 != None:
        nodes_specified = True
    boundary_GUs = node_list_by_attr(G, 'boundary_node', True)

    while True:  # breaks only when a valid cut line is created
        if n1 == None:
            n1 = rng.choice(boundary_GUs)
            node_prob = {}  # Dictionary that will contain the probability
                of node n2 being chosen. Nodes farther from n1 are more
                likely
        for n in boundary_GUs:
            node_prob[n] = spl[n1][n] ** 2 #probability is proportional to
                the square of the path length

        if n2 == None:
            n2 = rng.choices(boundary_GUs, node_prob.values(), k=1)[0]
        path = sp[n1][n2]
        G2 = G.copy()
        G2.remove_nodes_from(path)
        connected_comps = list(nx.connected_components(G2))
        r = rng.randint(0,1)
        #nodes_for_H = [None, None]
        #nodes_for_H[0] = connected_comps[0]
        #nodes_for_H[1] = connected_comps[1]
        #connected_comps[r] = G.subgraph(nodes_for_H[r])


        if len(connected_comps) == 2:  # Need *exactly* 2 connected
            components to break
            connected_comps[r].update(path) #Adds path to one of the
                subgraphs
            break
        else:
            if nodes_specified == True:
                raise ValueError("No valid cut line could be established
                    between nodes {n1} and {n2}.")
            n1 = None
            n2 = None
    return connected_comps

def crossover(G, ip, p0, p1, halves, gen, childid):
    """Takes two parent clusterings and creates a child by merging
        qualities from each parent
    G: Base graph
    ip: input
    p0: parent 0 clustering
    p1: parent 1 clustering
    halves: nodes from G that represent two halves of the graphs
    gen: generation we're on
    childid: labeled from 50 to 99. (assuming pop size 50) Will go here:
        stateG.nodes[n]['gen'][gen][childid] = (cl, dist)"""

    if p0.layer != 1 or p1.layer != 1: raise ValueError("parents must be
        clustering graphs")
    max_cl_num = 0
    for cl in p0.nodes:
        GU_nodes = p0.nodes[cl]['GUs']
        if(all(x in halves[0] for x in GU_nodes)):
            for gu in GU_nodes:
                G.nodes[gu]['gen'][gen][childid][0] = max_cl_num
            max_cl_num += 1

    for cl in p1.nodes:
        GU_nodes = p1.nodes[cl]['GUs']
        if(all(x in halves[1] for x in GU_nodes)):
            for gu in GU_nodes:
                G.nodes[gu]['gen'][gen][childid][0] = max_cl_num
            max_cl_num += 1

    unassigned_GUs = [n for n in G.nodes if
        G.nodes[n]['gen'][gen][childid][0] == -1]

    if len(halves[0]) < len(halves[1]):
        parents = [p0, p1]
    else:
        parents = [p1, p0]

    for p in parents:
        for cl in p.nodes:
            GU_nodes = p.nodes[cl]['GUs']
            if (all(x in unassigned_GUs for x in GU_nodes)):
                for gu in GU_nodes:
                    G.nodes[gu]['gen'][gen][childid][0] = max_cl_num
                max_cl_num += 1
                unassigned_GUs = list(set(unassigned_GUs) - set(GU_nodes))

    for gu in unassigned_GUs: #By default, all leftover nodes are
        singletons.
        G.nodes[gu]['gen'][gen][childid][0] = max_cl_num
        max_cl_num += 1
    ip.max_cl_num[(gen, childid)] = max_cl_num-1
```

```
def mutation(rng, G, ip, gen, pop_id):
    """mutates the graph based on the mutation probability"""
    mut_prob = ip.mutation_probability
    for n in G.nodes():
        mut_type = None
        n_cl = G.nodes[n]['gen'][gen][pop_id][0]  # The cluster number that
            n is in
        n_county = G.nodes[n]['county_num']
        r = rng.random()
        if r < mut_prob:  # If we hit that low probability
            nodes_in_n_cl = node_list_by_cluster(G, gen=gen, pop_id=pop_id,
                cl_val=n_cl)
            nodes_in_n_cl.remove(n)
            if len(nodes_in_n_cl) == 0:  # If n was in a singleton cluster
                already
                mut_type = 'merge'
            else:  # If n was not in a singleton cluster
                subgraph = nx.induced_subgraph(G, nodes_in_n_cl)
                # If removing n does not disconnect the current cluster
                if nx.is_connected(subgraph):
                    mut_type = rng.choice(['singleton', 'merge'])
        else:
            continue

        if mut_type == 'merge':  # We will randomly merge n into a
            neighboring cluster if possible
            nbrs = list(G.neighbors(n))
            rng.shuffle(nbrs)
            merge_success = False
            for nbr in nbrs:
                nbr_cl = G.nodes[nbr]['gen'][gen][pop_id][0] # The cluster
                    number for nbr
                nbr_county = G.nodes[nbr]['county_num']
                if n_cl != nbr_cl:
                    if (ip.metrics['cs'] == False) or (ip.metrics['cs'] ==
                        True and nbr_county == n_county):
                        G.nodes[n]['gen'][gen][pop_id][0] = nbr_cl
                            #Reassigns n's cluster number
                        merge_success = True
                        break
            if len(nodes_in_n_cl) == 0 and merge_success == True: # If n
                was in a singleton cluster before the merge
                nodes_in_max_cl = node_list_by_cluster(G, gen=gen,
                    pop_id=pop_id, cl_val=ip.max_cl_num[(gen, pop_id)])
                # The cluster number n_cl no longer exists in this
                    clustering, so we reassign the largest enumerated
                    cluster to n_cl
                for node in nodes_in_max_cl:
                    G.nodes[node]['gen'][gen][pop_id][0] = n_cl
                ip.max_cl_num[(gen, pop_id)] -= 1

        elif mut_type == 'singleton':  # We make node n become a singleton
            cluster
            new_cl_num = ip.max_cl_num[(gen, pop_id)] + 1
            G.nodes[n]['gen'][gen][pop_id][0] = new_cl_num
            ip.max_cl_num[(gen, pop_id)] += 1

def generate_plan(rng, G, ip, gen, cl_graph, pop_id):
    """Generates a districting plan based on the clustering
    G: base graph"""
    fpc = 0 #'failed plan count'
    revert=False
    while True: #Perpertual loop that will occur until a viable plan is
        created
        if fpc>0 and ip.randomness==0:
            revert=True
            ip.randomness=1
            used_centers, nbrlengths = generate_centers(rng, cl_graph,
                pop_id, ip, gen)
        else:
            used_centers, nbrlengths = generate_centers(rng, cl_graph,
                pop_id, ip, gen)
        # A dictionary that details which nodes are in a district
          (dist2node[dist] = [list of nodes])
        dist2node = {}
        for d in range(ip.num_districts): dist2node[d] = []
        # A dictionary containing the maximum size (in nodes) of each
          district
        maxsize = {}
        # A dictionary containing the current size (in nodes) of each
          district
        currsize = {}
        # A dictionary containing the cluster neighbors of the district
        dist_nbrs = {}

        unassigned_nodes = list(cl_graph[gen][pop_id].nodes)
        viable_dists = list(range(ip.num_districts))
        distpops = [0]*ip.num_districts
        for dist_num, center in enumerate(used_centers):
            maxsize[dist_num] = float('inf') # The districts should not be
                constrained by number of nodes
            currsize[dist_num] = 1
            distpops[dist_num] +=
                cl_graph[gen][pop_id].nodes[center]['POPULATION']
            dist_nbrs[dist_num] =
                set(cl_graph[gen][pop_id].neighbors(center))
            unassigned_nodes.remove(center)
            dist2node[dist_num].append(center)
            GUs_in_center = cl_graph[gen][pop_id].nodes[center]['GUs']
            for gu in GUs_in_center:
                # Assigns a distict number to the GU in the underlying graph
                G.nodes[gu]['gen'][gen][pop_id][1] = dist_num
            cl_graph[gen][pop_id].nodes[center]['districting'] = dist_num

        failed_plan = False
        while unassigned_nodes != []:
            if viable_dists == []: #If no districts can be grown and/or
                they are over 2*ideal_pop
                failed_plan = True
                fpc += 1
                clear_clustering(cl_graph[gen][pop_id], gen, pop_id,
                    stateG=G)
                print(f"Gen {gen}. Failed districting {pop_id} (no viable
                    districts).")
                break #Breaks from 'while unassigned_nodes != []' loop;
                    Restarts the 'while True' loop

            if ip.randomness==0: #deterministic district choices. Picks
                least populated viable district
                dist = -1
                distpops_copy = distpops.copy()
                while dist not in viable_dists:
                    if dist != -1:
                        distpops_copy[dist] = np.inf
                    dist = distpops_copy.index(min(distpops_copy))
            elif ip.randomness==1: #pseudo-random district choices
                #Picks a random viable district (weighted inversely
                    proportional to number of nodes in district)
                pop_weights = [total_pop(G, ip)-distpops[i] for i in
                    range(len(viable_dists))]
                mdw = min(pop_weights)
                pop_weights = [pop_weights[i] - mdw+1 for i in
                    range(len(pop_weights))]
                dist = rng.choices(viable_dists, weights=pop_weights,
                    k=1)[0]
            elif ip.randomness==2: #random district choices
                dist = rng.choice(viable_dists)
            if nbrlengths[dist] == {}: #Can occur if previous removal of GU
                from another district impacts this district
                viable_dists.remove(dist)
                continue

            currsize, maxsize, unassigned_nodes, dist2node, added_node,
                dist_nbrs, nbrlengths = grow_group(rng,
                cl_graph[gen][pop_id], ip, dist, gen, pop_id, currsize,
                maxsize, unassigned_nodes, dist2node, dist_nbrs,
                nbrlengths, viable_dists)
            if currsize[dist] == maxsize[dist]:
                # This occurs if dist cannot add any more clusters because
                    all surrounding clusters are assigned
                viable_dists.remove(dist)
            if added_node != None:
                GUs_in_added_node =
                    cl_graph[gen][pop_id].nodes[added_node]['GUs']
                for gu in GUs_in_added_node:
                    G.nodes[gu]['gen'][gen][pop_id][1] = dist
                distpops[dist] +=
                    cl_graph[gen][pop_id].nodes[added_node]['POPULATION']
                ideal_pop = target_pop(G, ip)
                # If a district has exceeded twice the ideal population,
                    stop adding clusters to it
                if distpops[dist] > 2*ideal_pop*ip.alpha**gen:
                    #ip.alpha**gen is a modifier that slowly reduces this
                    UB to tighten population requirements
```

```
                    try: viable_dists.remove(dist)
                    except ValueError: pass #May have already removed dist
                        from viable_dists in currsize == maxsize check


        if any(distpops[i] < 0.15*target_pop(G, ip) for i in
            range(ip.num_districts)) and failed_plan == False: #'and'
            present so we clear clustering only once
            failed_plan = True
            clear_clustering(cl_graph[gen][pop_id], gen, pop_id, stateG=G)
            print(f"Gen {gen}. Failed districting {pop_id} (underpopulated
                district).")

        if failed_plan == False:
            break  # breaks from 'while True' loop because a valid plan was
                produced
    if revert==True: ip.randomness=0 #Reverts the randomness setting back
        to 0 if it was changed
    return cl_graph

def wilson(rng, graph):
    '''Returns a uniform spanning tree on G'''
    walk = loopErasedWalk(rng, graph)
    currentNodes = [n for n in walk]

    uniformTree = nx.Graph()
    for i in range(len(walk) - 1):
        uniformTree.add_edge(walk[i], walk[i + 1])

    treeNodes = set(uniformTree.nodes)
    neededNodes = set(graph.nodes) - treeNodes

    while neededNodes:
        v = rng.choice(sorted(list(neededNodes))) # sort for code
            repeatability
        walk = loopErasedWalk(rng, graph, v1 = [v], v2 = currentNodes)
        currentNodes += walk
        for i in range(len(walk) - 1):
            uniformTree.add_edge(walk[i], walk[i + 1])
        treeNodes = set(uniformTree.nodes)
        neededNodes = set(graph.nodes) - treeNodes

    pass
    nx.set_node_attributes(uniformTree, dict(graph.nodes("POPULATION")),
        "POPULATION")
    nx.set_node_attributes(uniformTree, dict(graph.nodes("area")), "area")
    nx.set_node_attributes(uniformTree, dict(graph.nodes("PresBlue")),
        "PresBlue")
    nx.set_node_attributes(uniformTree, dict(graph.nodes("PresRed")),
        "PresRed")
    nx.set_edge_attributes(uniformTree, dict(graph.edges("shared_perim")),
        "shared_perim")
    nx.set_edge_attributes(uniformTree, dict(graph.edges("Dist_Boundary")),
        "Dist_Boundary")
    return uniformTree
```

```
def loopErasedWalk(rng, graph, v1 = None, v2 = None):
    '''Returns a loop-erased random walk between components v1 & v2'''
    if v1 is None:
        v1 = [rng.choice(sorted(list(graph.nodes)))]
    if v2 is None:
        v2 = [rng.choice(sorted(list(graph.nodes)))]

    v = rng.choice(sorted(v1))
    walk = [v]
    while v not in v2:
        v = rng.choice(sorted(list(graph.neighbors(v))))
        if v in walk:
            walk = walk[0:walk.index(v)]
        walk.append(v)

    return walk

def find_edge_cut(rng, tree, tol):
    '''Input a tree graph and a percent tolerance. The function will remove
        a
    random edge that splits the tree into two pieces such that each piece
    has population within that percent tolerance. The variable 'tol' should
        be a positive real
    number in (0,100].'''
    if tol > 100 or tol <= 0 or (isinstance(tol, float) == False and
        isinstance(tol, int) == False):
        raise ValueError("tol must be a float or integer variable in the
            range (0,100].")
    if nx.is_tree(tree) == False:
        raise ValueError("The input graph must be a tree.")
    tree_edge_list = list(tree.edges)
    rng.shuffle(tree_edge_list)  #Randomly shuffles the edges of T
    e = None
    num_edges = len(tree_edge_list)

    for i in range(num_edges):
        e = tree_edge_list[i]  #Edge to delete
        tree.remove_edge(*e)
        subgraphs = nx.connected_components(tree)
        subgraphs_lst = list(subgraphs)
        subgraphs_lst[0] = sorted(subgraphs_lst[0])
        subgraphs_lst[1] = sorted(subgraphs_lst[1])
        dist_pop1 = sum(value for key, value in
            nx.get_node_attributes(tree, "POPULATION").items() if key in
            subgraphs_lst[0])  #Finds population sum for first district
        dist_pop2 = sum(value for key, value in
            nx.get_node_attributes(tree, "POPULATION").items() if key in
            subgraphs_lst[1])  #Finds population sum for second district
        total_pop = dist_pop1 + dist_pop2
        avg_pop = total_pop / 2
        if abs(dist_pop1-avg_pop) > 0.01*tol*avg_pop or
            abs(dist_pop2-avg_pop) > 0.01*tol*avg_pop:  #If both proposed
            districts are outside the prescribed tolerance
            tree.add_edge(*e)  #Adds the edge back to the tree if it didn't
                meet the tolerance
        else:  #This is what we want: both proposed districts within the
            prescribed tolerance
            if i == 0:
                #print(f"Population requirement was met. Removing edge {e}.
                    Required {i+1} iteration.")
                pass
            else:
                #print(f"Population requirement was met. Removing edge {e}.
                    Required {i+1} iterations.")
                pass
            return subgraphs_lst
        if i == num_edges - 1:
            #print(f"No subgraphs with appropriate criteria requirements
                were found. Required {i+1} iterations.")
            return []  #Returns empty subgraphs list if no appropriate
                subgraphs were found.

def generate_centers(rng, cl_graph, pop_id, ip, gen):
    """Determines which clusters we should center the districts around"""
    if ip.randomness == 2: #completely random centers
        used_centers = rng.sample(list(cl_graph[gen][pop_id].nodes),
            ip.num_districts)
    elif ip.randomness == 1: #centers chosen that are far apart
        num_sets_centers = 5
        centers = [None]*num_sets_centers
        summed_center_length = [0 for _ in range(num_sets_centers)]
        for j in range(num_sets_centers): #Randomly selects n clusters
            which will form the base of our districts
            centers[j] = rng.sample(list(cl_graph[gen][pop_id].nodes),
                ip.num_districts)
            for c_idx, c in enumerate(centers[j]):
                for d_idx, d in enumerate(centers[j]):
                    #We find the summed graph distance between all centers
                    if c_idx < d_idx: summed_center_length[j] +=
                        nx.shortest_path_length(cl_graph[gen][pop_id],
                        source=c, target=d)
        used_centers_idx =
            summed_center_length.index(max(summed_center_length)) #Finds
            the index for the centers set with most dispersion
        used_centers = centers[used_centers_idx]

    elif ip.randomness == 0: #deterministic centers chosen by highest
        population that are far enough apart
        pop_dict = dict(cl_graph[gen][pop_id].nodes("POPULATION"))
        sorted_pop_dict = dict(sorted(pop_dict.items(), key=lambda item:
            item[1], reverse=True))
        cl_list = list(sorted_pop_dict.keys())
        used_centers = []
        aspl = nx.average_shortest_path_length(cl_graph[gen][pop_id])
        min_distance = aspl*0.8 #minimum mutual distance that the nodes
            must be separated by
        k=0
        while len(used_centers) < ip.num_districts: #Will select the
            [ip.num_districts] centers that are highest populated *and* at
            least a certain distance apart from each other
            if len(used_centers) == 0: used_centers.append(cl_list[k])
            elif k<len(cl_list)-1: #If the candidate center is sufficiently
                far from all current centers, add it to the list
                k+=1
                #if min([nx.shortest_path_length(cl_graph[gen][pop_id],
                    source=cl_list[k], target=cent) for cent in
                    used_centers]) > diameter/math.sqrt(ip.num_districts):
                if min([nx.shortest_path_length(cl_graph[gen][pop_id],
                    source=cl_list[k], target=cent) for cent in
                    used_centers]) > min_distance:
                    used_centers.append(cl_list[k])
            elif k == len(cl_list)-1: #We have explored all clusters and
                cannot find centers that are sufficiently far apart
                min_distance = min_distance*0.8
                print(f"reducing minimum distance requirement. New min dist
                    = {min_distance}")
                used_centers = []
                k=0

    #Starts a dictionary that records nbrlengths for each district
    nbrlengths = {} #nbrlengths[dist][nbr] = length_with_dist
    for dist, c in enumerate(used_centers):
        nbrlengths[dist] = {}
        for nbr in cl_graph[gen][pop_id].neighbors(c):
            if cl_graph[gen][pop_id].nodes[nbr]['districting'] == -1:
                nbrlengths[dist][nbr] =
                    cl_graph[gen][pop_id].edges[(c,nbr)]['shared_perim']
        if ip.randomness <= 1: nbrlengths[dist] =
            dict(sorted(nbrlengths[dist].items(), key=lambda x:x[1],
            reverse=True))
    return used_centers, nbrlengths

def checkmetrics(dg, ip):
    """Checks whether the metrics of distgraph are within acceptable
        parameters"""
    if ip.metrics['dpop']:
        if dg.pop_dev_sum > ip.ub["dpop"]: #Acceptable pop_dev allows an
            average district to be 40% off from ideal
            return False, 'dpop', dg.pop_dev_sum
    if ip.metrics['comp']:
        #if dg.shifted_pp_comp > 0.8888: #Acceptable average compactness is
            less than 0.8888
        if dg.shifted_pp_comp > ip.ub["comp"]:
            return False, 'comp', dg.shifted_pp_comp
    if ip.metrics['eg']:
        if dg.eg > ip.ub["eg"]: #Acceptable efficiency gaps are 3*8% (>8%
            is considered problematic by Stephanopoulos and McGhee)
            return False, 'eg', dg.eg
    if ip.metrics['mm']:
        if dg.mm > ip.ub["mm"]: #Acceptable median-mean calculations allow
            <= 0.05
            return False, 'mm', dg.mm
    if ip.metrics['cs']:
        if dg.cs > ip.ub["cs"]: #Acceptable county-split count is 50 or
            fewer
            return False, 'cs', dg. cs
    if ip.metrics['egu']:
        if dg.egu > ip.ub["egu"]: #Acceptable excess gu count is 500 or
            fewer
            return False, 'egu', dg.egu
    #If we've made it this far, then all metrics are okay.
    return True, ":)", math.pi


def dominates(solution1, solution2):
    """Check if solution1 dominates solution2. 'solution1' and 'solution2'
       are graphs"""
    s1_mlist = solution1.metrics
    s2_mlist = solution2.metrics
    doms = all(s1 <= s2 for s1, s2 in zip(s1_mlist, s2_mlist)) and any(s1 <
       s2 for s1, s2 in zip(s1_mlist, s2_mlist))
    return doms

def nondominated_sorting(dist_graph_list):
    """Perform nondominated sorting on a dg_list of solutions."""
    # Initialize ranks, dominated solutions, and dominated-by count
    dgl = dist_graph_list #An alias
    ranks = [0] * len(dgl)
    dominated_solutions = [[] for _ in range(len(dgl))]
    dominated_by_count = [0] * len(dgl)

    # Determine dominated solutions and count dominated-by for each solution
    for i, dg1 in enumerate(dgl):
        for j, dg2 in enumerate(dgl):
            if i < j:
                if dominates(dg1, dg2):
                    # dg1 dominates dg2
                    dominated_solutions[i].append(j)
                    dominated_by_count[j] += 1
                elif dominates(dg2, dg1):
                    # dg2 dominates dg1
                    dominated_by_count[i] += 1
                    dominated_solutions[j].append(i)

        if dominated_by_count[i] == 0:
            # No solutions dominate solution1, it belongs to the first rank
            ranks[i] = 1

    # Assign solutions to different fronts based on dominance relationships
    front = 1
    while any(rank == front for rank in ranks):
        next_front = []
        for i, rank in enumerate(ranks):
            if rank == front:
                for j in dominated_solutions[i]:
                    dominated_by_count[j] -= 1
                    if dominated_by_count[j] == 0:
                        # No solutions dominate solution j, it belongs to
                            the next front
                        ranks[j] = front + 1
                        next_front.append(j)
        front += 1

    # Create the fronts based on the ranks
    fronts_idx = [[] for _ in range(front-1)] #records the index of the
        plan in the given front
    fronts = [[] for _ in range(front-1)] #records the fronts themselves
    for i, rank in enumerate(ranks):
        fronts_idx[rank - 1].append(i)
        fronts[rank - 1].append(dgl[i])
        if isinstance(dgl[i], nx.classes.graph.Graph) == False: raise
            ValueError("dgl[i] must be a graph")
    return fronts_idx, fronts

def nondominated_sorting2(dist_graph_list):
    """Sorts the entries of dist_graph_list into fronts"""
    dgl = dist_graph_list #An alias
    dominated_solns = [] #This will contain the dominated plans.
    nondom = [] #This will contain the nondominated plans.
    nondom_idxs = [] #This will contain the indices of the nondominated
        plans.
    rank=0 #This will denote the front the plan is assigned to
    fronts = [] #List of lists. Each entry will be the list of plans in
        that front
    fronts_idxs = [] #List of lists. Each entry will the list of plan
        indices in that front
    plan_ids = {}
    for i,dg in enumerate(dgl): plan_ids[dg] = i #This dictionary assigns
        each dist_graph dg to an index i
    #Loop for first front
    for dg1 in dgl:
        for dg2 in dgl:
            if dominates(dg2, dg1): #dg2 dominates dg1
                dominated_solns.append(dg1)
                break #no more calculations in second 'for' loop are needed
                    since we've determined that dg1 is dominated

    nondom = list(set(dgl) - set(dominated_solns))
    fronts.append(nondom)
    for dg in nondom: nondom_idxs.append(plan_ids[dg])
    fronts_idxs.append(nondom_idxs)

    #Loop for every subsequent front
    while dominated_solns != []:
        rank += 1 #helpful for debugging
        temp_domsols = []
        nondom_idxs = [] #This needs to reset every loop
        for dg1 in dominated_solns:
            for dg2 in dominated_solns:
                if dominates(dg2, dg1): #dg2 dominates dg1
                    temp_domsols.append(dg1)
                    break
        nondom = list(set(dominated_solns) - set(temp_domsols))
        fronts.append(nondom)
        for dg in nondom: nondom_idxs.append(plan_ids[dg])
        fronts_idxs.append(nondom_idxs)
        dominated_solns = temp_domsols #makes a (shallow) copy
        del temp_domsols #Deletes the temporary variable
    return fronts_idxs, fronts


def crowding_distance(front, front_pop_id, num_to_keep, min_pop_id):
    """Finds the crowding distance for each solution in a given front and
    returns the least crowded solutions up to the num_to_keep"""
    cd = {}  # Crowding distance
    for idx, _ in enumerate(front):
        cd[idx] = 0
    num_metrics = len(front[0].metrics)
    for i in range(num_metrics):  # 'i' represents the metric index
        soln_list = sorted([(idx, soln.metrics[i]) for idx, soln in
            enumerate(front)], key=lambda x: x[1])  # sorts metric list by
            metric value
        # maxval - minval finds range for the metric
        range_metric = soln_list[-1][1] - soln_list[0][1]
        # 'j' will be used to identify spot in list
        for j, soln in enumerate(soln_list):
            idx = soln[0]  # 'idx' represents the index of the solution
            if j == 0 or j == len(soln_list)-1:
                cd[idx] += float('inf')
            else:
                try:
                    cd[idx] += (soln_list[j+1][1] - soln_list[j-1][1]) /
                        range_metric
                except ZeroDivisionError:
                    cd[idx] += (soln_list[j+1][1] - soln_list[j-1][1]) / 1
    sorted_idxs = sorted([(idx, val) for idx, val in cd.items()],
        key=lambda x: x[1], reverse=True)
    sorted_front = [front[i[0]] for i in sorted_idxs]
    sorted_pop_ids = [front_pop_id[i[0]] for i in sorted_idxs]
    return sorted_pop_ids, sorted_front[0:num_to_keep]  # Keeps the best
        'num_to_keep' graphs

def assign_next_gen(G, ip, gen, front, front_pop_id, cl_graph_front,
    dist_graph, cl_graph, min_pop_id):
    """Assigns the GUs of G to the proper clusterings and districts in the
        provided generation"""
    for idx, dg in enumerate(front): #dg = dist_graph
        pop_id = min_pop_id + idx
        if ip.metrics['dpop']: G.pop_dev[gen][pop_id] = dg.pop_dev_sum
        if ip.metrics['comp']: G.comp[gen][pop_id] = dg.shifted_pp_comp
        if ip.metrics['eg']: G.eg[gen][pop_id] = dg.eg
        if ip.metrics['mm']: G.mm[gen][pop_id] = dg.mm
        if ip.metrics['cs']: G.cs[gen][pop_id] = dg.cs
        if ip.metrics['egu']: G.egu[gen][pop_id] = dg.egu
        dist_graph[gen][pop_id] = dg
        cl_graph[gen][pop_id] = cl_graph_front[idx]
        ip.max_cl_num[(gen, pop_id)] = ip.max_cl_num[(gen-1,
            front_pop_id[idx])]

        node_count = 0
        for dist in dg.nodes:
            for cl in dg.nodes[dist]['clgu']:
                for gu in dg.nodes[dist]['clgu'][cl]:
                    G.nodes[gu]['gen'][gen][pop_id] = (cl, dist)
                    node_count+=1
        if node_count != ip.num_GUs:
            raise ValueError(f"We should have {ip.num_GUs} GUs")

def plot_plans(G, county_lines, min_idx=0, max_idx=None):
    """Plots all maps between min_idx and max_idx"""
    high_contrast_colors = ['blue', 'green', 'yellow', 'magenta', 'cyan',
        'orange', 'purple']
    cmap = mcolors.ListedColormap(high_contrast_colors)
    if max_idx == None:
        i=min_idx
        planx = 'plan' + f'{i}'
        while planx in G.data:
            fig, ax = plt.subplots(figsize=(10,10))
            G.data.plot(column=planx, ax=ax, legend=True, cmap=cmap)
            county_lines.data.boundary.plot(ax=ax, color="black")
            ax.set_title(f"Plan {i}")
            fig.show()
            i += 1
            planx = 'plan' + f'{i}'
    else:
        for i in range(min_idx, max_idx):
            planx = 'plan' + f'{i}'
            fig, ax = plt.subplots(figsize=(10,10))
            G.data.plot(column=planx, ax=ax, legend=True, cmap=cmap)
            county_lines.data.boundary.plot(ax=ax,color="black")
            ax.set_title(f"Plan {i}")
            fig.show()

def plot_plan(G, county_lines, idx):
    """Plots one plan with county lines"""
    high_contrast_colors = ['blue', 'green', 'yellow', 'magenta', 'cyan',
        'orange', 'purple']
    cmap = mcolors.ListedColormap(high_contrast_colors)
    planx = 'plan' + f'{idx}'
    fig, ax = plt.subplots(figsize=(10,10))
    G.data.plot(column=planx, ax=ax, legend=True, cmap=cmap)
    county_lines.data.boundary.plot(ax=ax, color="black")
    fig.show()

def plot_front(fronts, ip):
    """Plots all fronts. Requires exactly two metrics"""
    if len(fronts[0][0].metrics) != 2: raise ValueError("Can only plot
        exactly 2 metrics")
    color_list = ['b', 'r', 'g', 'c', 'm', 'y', 'k']
    num_fronts = len(fronts)
    xvals = [None] * num_fronts
    yvals = [None] * num_fronts
    for idx, front in enumerate(fronts):
        xvals[idx] = [p.metrics[0] for p in front]
        yvals[idx] = [p.metrics[1] for p in front]
        plt.plot(xvals[idx], yvals[idx], color=color_list[idx%7],
            marker='o', linestyle='none', label=f'front {idx}')

    metric_names = [k for k in ip.metrics.keys() if ip.metrics[k] == True]
    plt.xlabel(metric_names[0])
    plt.ylabel(metric_names[1])
    plt.title("Fronts")
    plt.legend()
    plt.show()

def plot_hvolume(hvolume):
    """Plots the hypervolume as a line chart.
    hvolume: list of hypervolume values by generation."""
    # Create a figure and a set of subplots
    fig, ax = plt.subplots(nrows=1, ncols=1)

    # Plot on the first subplot
    ax.plot(hvolume)

    # Add a title to the subplot
    ax.set_title("Hypervolume by Generation")

    # Add labels to the axes
    ax.set_xlabel("Generation")
    ax.set_ylabel("Hypervolume")

    # Set x-axis ticks to be integers
    ax.set_xticks(range(len(hvolume)))

    # Display the chart
    fig.show()

def radar_plot(last_gen, ip, idx):
    """Creates a radar plot for the metrics in the given plan (defined by
       idx)"""
    theta = [ip.long_met_names[met] for met in list(ip.metrics.keys()) if
       ip.metrics[met]==True]
    max_met_list = list(last_gen.max())[2:] #Excludes the Generation and
       pop_id cols
    upd_idx = idx + 2*ip.num_generations*ip.pop_size
    met_list = list(last_gen.loc[upd_idx])[2:]
    dict_for_radar = {}
    dict_for_radar['theta'] = theta
    dict_for_radar['r'] = [met_list[i]/max_met_list[i] for i in
       range(len(met_list))]

    df_for_radar = pd.DataFrame(dict_for_radar)
    fig = px.line_polar(df_for_radar, r='r', theta='theta',
       line_close=True, range_r=[0,1])
    fig.update_traces(fill='toself')
    fig.show()
```

```
def check_district_contiguity(G, ip, gen, cl_graph, min_pop_id=0,
    max_pop_id=None):
    """Checks that the plan will be have a contiguous districting"""
    if max_pop_id == None: max_pop_id = 2*ip.pop_size
    for pop_id in range(min_pop_id, max_pop_id):
        for dist in range(ip.num_districts):
            nodes_in_dist = node_list_by_district(G, gen, pop_id, dist)
            subgraph = nx.induced_subgraph(G, nodes_in_dist)
            if nx.is_connected(subgraph) == False:
                raise RuntimeError("All districts should be contiguous")
        print(f"Gen {gen}, pop_id {pop_id}. All districts are contiguous")

def check_cluster_contiguity(G, ip, gen, cl_graph, min_pop_id=0,
    max_pop_id=None):
    """Checks that the plan will be have a contiguous clustering"""
    if max_pop_id == None: max_pop_id = 2*ip.pop_size
    for pop_id in range(min_pop_id, max_pop_id):
        for cl in range(ip.max_cl_num[(gen, pop_id)]):
            nodes_in_cl = node_list_by_cluster(G, gen, pop_id, cl)
            subgraph = nx.induced_subgraph(G, nodes_in_cl)
            if nx.is_connected(subgraph) == False:
                raise RuntimeError("All clusters should be contiguous")
        print(f"Gen {gen}, pop_id {pop_id}. All clusters are contiguous")

def hypervolume(ip, gen_col2, front_col, metlist):
    """Calculates the hypervolume columns of values given a graph G
    metlist[0] will contain all scaled pop_dev values; metlist[1] might
        contain all scaled
    compactness values"""
    front0 = [[] for _ in range(ip.num_generations+1)]
    #front0 = np.empty(shape=(0, ip.num_generations))
    volume = []
    ref = [1.1 for _ in range(len(metlist))]
    for row in range(len(metlist[0])):
        gen = gen_col2[row]
        if front_col[row] == 0: #If the entry is in rank 0
            front0[gen].append([metlist[i][row] for i in
                range(len(metlist))])
            #front0[gen] = [metlist[i][row] for i in range(len(metlist))]
    for gen in range(len(front0)):
        front0[gen] = np.array(front0[gen])
    #front0arr = np.array(front0)
    minval = [[] for _ in range(len(metlist))] #minval[gen][metric] = number
    for gen in range(ip.num_generations+1):
        #hypvol = hv.HyperVolume(ref)
        #volume.append(hypvol.compute(front0[gen]))
        ind = HV(ref_point=ref)
        volume.append(ind(front0[gen]))
    return volume
```

```
def main(*args):
    timetxt = datetime.datetime.now().strftime("_%m%d%y_%H%M%S_%f")
    rng = random.Random(args[1]) #seeds random

    if isinstance(args[0], str) == True:
        file_name = args[0]
        ip = input_c.Load_from_file(file_name)
    elif args[0] == None:
        print("No input file provided. Running with default parameters")
        ip = input_c(
            json_file="./SC_Precincts_2_FeaturesToJSO.geojson",
            ps=50,  # population size
            mp=0.05,  # mutation probability
            nd=7,  # number of districts
            ng=3,  # number of generations
            met={  # metrics
                "dpop": True,  # population
                "comp": True,  # compactness
                "eg": True,  # efficiency gap
                "mm": False,  # median-mean
                "cs": True,  # county splits
                "egu": True # excess geographic units
            },
            cf="./SC_Counties_20_FeaturesToJSO.geojson",
            lmn={  # long metric names
                "dpop": "Pop Dev",  # population
                "comp": "Compactness",  # compactness
                "eg": "Eff Gap",  # efficiency gap
                "mm": "Med Mean",  # median-mean
                "cs": "County Splits",  # county splits
                "egu": "Excess GUs" # excess geographic units
            },
            rnd=1 #Randomness; 0: deterministic, 1: some randomness, 2:
                total randomness
        )
    ip.rand_seed = args[1]
    ip.ub = {} # Upper bounds for the various metrics
    ip.ub["dpop"] = None # Initialize this in the next step
    ip.ub["comp"] = 0.90909 #Corresponds to invPP of 10
    ip.ub["eg"] = 0.24
    ip.ub["mm"] = 0.05
    ip.ub["cs"] = 50
    ip.ub["egu"] = 500
    gen = 0
    start = time.time()
```

```
# ------------------------------------------------------------
# Initialize Graph
stateG = initialize_graph(ip)
ip.ub["dpop"] = ip.num_districts*target_pop(stateG, ip)*0.4
print("Initialized graph")
cp0 = time.time()

# ------------------------------------------------------------
# Create attribute graph (for counties)
countygraph = create_attr_graph(stateG, ip, attr="county_num")
print("Finished county graph")
ip.num_counties = countygraph.number_of_nodes()
cp1 = time.time()

# ------------------------------------------------------------
# Create shortest paths dictionary
try:
    with open('sp.pkl', 'rb') as fp: shortest_paths = pickle.load(fp)
except FileNotFoundError:  # If we haven't done this before
    shortest_paths = nx.shortest_path(stateG)
    with open('sp.pkl', 'wb') as fp: pickle.dump(shortest_paths, fp)
print("Created sp dictionary")
cp2 = time.time()

# ------------------------------------------------------------
# Create shortest path lengths dictionary
try:
    with open('spl.pkl', 'rb') as fp: spl = pickle.load(fp)
except FileNotFoundError:  # If we haven't done this before
    spl = np.zeros([len(shortest_paths), len(shortest_paths)],
        dtype=int)  # Shortest path lengths
    for i in range(len(shortest_paths)):
        for j in range(len(shortest_paths)):
            spl[i][j] = len(shortest_paths[i][j])
    with open('spl.pkl', 'wb') as fp: pickle.dump(spl, fp)
print("Created spl dictionary")
cp3 = time.time()

# ------------------------------------------------------------
# Initialize clusterings
for i in range(ip.pop_size):
    initialize_clustering(random, stateG, ip, i)
print("Initialized clusterings")
cp4 = time.time()


# ------------------------------------------------------------
# Create cluster graphs
#Create empty dist_graph and cl_graph array. dist_graph[gen][pop_id]
   gives the correct graph
dist_graph = [[None for _ in range(ip.pop_size*2)] for _ in
   range(ip.num_generations+1)]
cl_graph = [[None for _ in range(ip.pop_size*2)] for _ in
   range(ip.num_generations+1)]
for pop_id in range(ip.pop_size):
    cl_graph[gen][pop_id] = create_attr_graph(stateG, ip,
        gen_id_cldi=(gen, pop_id, 0))
    cl_graph[gen][pop_id].mut=False #Indicates no mutation has been
        applied
print("Created initial cluster graphs")
cp5 = time.time()

# ----------------------------------------------------------------
# Initialize Time variables
time_cut_line = 0
time_crossover = 0
time_mutation = 0
time_cluster_g = 0
time_gen_plans = 0
time_nondom_sort = 0
time_crowd_dist = 0

# ----------------------------------------------------------------
# Initialize front_col2 (This contains the rank for each map)
front_col2 = [None]*(ip.num_generations+1)*ip.pop_size*2

# ----------------------------------------------------------------
# Population modifier
ip.alpha = (1.3/2) ** (1/ip.num_generations) #Population modifier --
   will provide an upper bound on district population

while gen < ip.num_generations:
    cp5_ = time.time()
    # ----------------------------------------------------------------
    # Create cut lines
    graph_halves = [None]*ip.pop_size
    for i in range(ip.pop_size):
        graph_halves[i] = create_cut_line(random, stateG,
            shortest_paths, spl)
    cp6 = time.time()
    time_cut_line += cp6-cp5_

    # ----------------------------------------------------------------
    # Crossover
    for i in range(ip.pop_size):
        [p0, p1] = rng.sample(cl_graph[gen][0:ip.pop_size], 2) #Selects
            two random cluster graphs from generation 'gen'
        childid = i + ip.pop_size
        crossover(stateG, ip, p0, p1, graph_halves[i], gen, childid)
    cp7 = time.time()
    time_crossover += cp7-cp6

    # ----------------------------------------------------------------
    # Mutation
    muts = round(ip.mutation_probability*ip.pop_size)  # Number of
        mutations to perform
    if muts == 0: muts = 1 #Round up to 1 if pop size is too small
    ids = list(range(ip.pop_size, 2*ip.pop_size-1)) #Can only mutate
        child graphs
    map_ids_to_mut = rng.choices(ids, k=muts)
    #print(map_ids_to_mut)
    for child_id in map_ids_to_mut:
        mutation(random, stateG, ip, gen, child_id)
    cp8 = time.time()
    time_mutation += cp8-cp7

    # ----------------------------------------------------------------
    # Create children cluster graphs
    for pop_id in range(ip.pop_size, 2*ip.pop_size):
        cl_graph[gen][pop_id] = create_attr_graph(stateG, ip,
            gen_id_cldi=(gen, pop_id, 0))
        cl_graph[gen][pop_id].mut = False
    for map_id in map_ids_to_mut:
        cl_graph[gen][map_id].mut = True #Indicates that these were
            mutated using 'merge' and 'singleton' and will be mutated
            using recom
    cp9 = time.time()
    time_cluster_g += cp9-cp8

    # ----------------------------------------------------------------
    # Generate Plans
    num_plans_to_make = 2*ip.pop_size if gen==0 else ip.pop_size

    for i in range(num_plans_to_make):
        if gen==0: pop_id = i
        else: pop_id = i+ip.pop_size
        viableplan = False
        fpc = 0 #failed plan count
        while viableplan == False: #viable plan will be True when the
            metrics are within acceptable parameters
            if fpc > 20: cl_graph[gen][pop_id].mut = False
            if fpc > 50: # If 50 iterations were insufficient to find a
                viable plan, copy the previous one.
                cl_graph[gen][pop_id] =
                    copy.deepcopy(cl_graph[gen][pop_id-1])
                dist_graph[gen][pop_id] =
                    copy.deepcopy(dist_graph[gen][pop_id-1])
                viableplan, failedmet, metval =
                    checkmetrics(dist_graph[gen][pop_id], ip)
                if viableplan == False: raise RuntimeError("viableplan
                    shouldn't be false after copying previous plan.")
                print(f"Gen {gen}. Failed to create plan {pop_id} 50
                    times. Copying plan {pop_id-1}.")
                break #Breaks from 'while viableplan == False:' loop
            if fpc > 0 and ip.randomness == 0: #temporarily changes the
                randomness setting to create a map
                ip.randomness = 1
                cl_graph = generate_plan(random, stateG, ip, gen,
                    cl_graph, pop_id)
                ip.randomness = 0
            else:
                cl_graph = generate_plan(random, stateG, ip, gen,
                    cl_graph, pop_id)
            dist_graph[gen][pop_id] =
                create_attr_graph(cl_graph[gen][pop_id], ip,
            attr='districting')
        if cl_graph[gen][pop_id].mut: #Mutates graph by
            recombination of two districts
            while True:
                d1 = rng.randint(0,ip.num_districts-1)
                d2 = rng.choice(
                    list(dist_graph[gen][pop_id].neighbors(d1)))
                d1_pop =
                    dist_graph[gen][pop_id].nodes[d1]["POPULATION"]
                d2_pop =
                    dist_graph[gen][pop_id].nodes[d2]["POPULATION"]
                if np.sign(
                    d1_pop-ip.ideal_pop)*np.sign(d2_pop-ip.ideal_pop)
                    <= 0: break #breaks only if dists are on
                    opposite sides of the population balance line
            #print(d1, d2)
            cl_nodes_to_recom =
                list(dist_graph[gen][pop_id].nodes[d1]['clgu']) +
                list(dist_graph[gen][pop_id].nodes[d2]['clgu'])
            subg = nx.induced_subgraph(cl_graph[gen][pop_id],
                cl_nodes_to_recom)
            tree = wilson(random, subg)
            recom_dists = find_edge_cut(random, tree, 5) # 5%
                tolerance
            for idx, dist in enumerate(recom_dists):
                if idx==0: d = d1
                elif idx==1: d = d2
                else: raise ValueError("Should only be two
                    districts after recom")
                for cl in dist:
                    cl_graph[gen][pop_id].nodes[cl]['districting']
                        = d #assigns clusters to new districts in
                        cl_graph
                    for gu in
                        cl_graph[gen][pop_id].nodes[cl]['GUs']:
                         stateG.nodes[gu]['gen'][gen][pop_id][1] = d
                             #assigns GUs to districts in base graph
            dist_graph[gen][pop_id] =
                create_attr_graph(cl_graph[gen][pop_id], ip,
                attr='districting') #Recreates the dist_graph
        viableplan, failedmet, metval =
            checkmetrics(dist_graph[gen][pop_id], ip)
        if viableplan == False:
            print(f"Gen {gen}. Failed districting {pop_id} (metric
                too large: {failedmet}={metval}).")
            fpc += 1
            clear_clustering(cl_graph[gen][pop_id], gen, pop_id,
                stateG=stateG)
        else:
            print(f"Gen {gen}. Made it through districting
                {pop_id}.")
    # Record the metrics
    if ip.metrics['dpop']: stateG.pop_dev[gen][pop_id] =
        dist_graph[gen][pop_id].pop_dev_sum
    if ip.metrics['comp']: stateG.comp[gen][pop_id] =
        dist_graph[gen][pop_id].shifted_pp_comp
    if ip.metrics['eg']: stateG.eg[gen][pop_id] =
        dist_graph[gen][pop_id].eg
    if ip.metrics['mm']: stateG.mm[gen][pop_id] =
        dist_graph[gen][pop_id].mm
    if ip.metrics['cs']: stateG.cs[gen][pop_id] =
        dist_graph[gen][pop_id].cs
    if ip.metrics['egu']: stateG.egu[gen][pop_id] =
        dist_graph[gen][pop_id].egu
    stateG.metrics[gen][pop_id] = dist_graph[gen][pop_id].metrics

cp10 = time.time()
time_gen_plans += cp10-cp9

# -----------------------------------------------------------------
# Nondominated sorting
fronts_pop_id, fronts = nondominated_sorting2(dist_graph[gen])
cl_graph_fronts = [[cl_graph[gen][i] for i in lst] for lst in
    fronts_pop_id]
cp11 = time.time()
time_nondom_sort += cp11-cp10


for rank, f in enumerate(fronts_pop_id):
    for pop_id in f:
        front_col2[gen*ip.pop_size*2 + pop_id] = rank #Assigns the
            rank (i.e. the front number) that each plan falls in

# -----------------------------------------------------------------
# Crowding Distance and Assigning next generation
front_lengths = [len(f) for f in fronts_pop_id]
stateG.front0_sizes.append(front_lengths[0])
cum_lengths = [sum(front_lengths[:i+1]) for i in
    range(len(front_lengths))]
if gen < ip.num_generations-1:
    for idx, l in enumerate(cum_lengths):
        if idx == 0: min_pop_id=0
        else: min_pop_id=cum_lengths[idx-1]

        if l <= ip.pop_size:
            assign_next_gen(stateG, ip, gen+1, fronts[idx],
                fronts_pop_id[idx], cl_graph_fronts[idx],
                dist_graph, cl_graph, min_pop_id)
        elif l > ip.pop_size:
            num_to_keep = front_lengths[idx] - cum_lengths[idx] +
                ip.pop_size
            pf_pop_ids, partial_front =
                crowding_distance(fronts[idx], fronts_pop_id[idx],
                num_to_keep, min_pop_id)
            assign_next_gen(stateG, ip, gen+1, partial_front,
                pf_pop_ids, cl_graph_fronts[idx], dist_graph,
                cl_graph, min_pop_id)
            break #Don't assign any more to the next gen
else: #If we are on the last generation, only assign the Pareto
    front
    assign_next_gen(stateG, ip, gen+1, fronts[0], fronts_pop_id[0],
        cl_graph_fronts[0], dist_graph, cl_graph, min_pop_id=0)
cp12 = time.time()
time_crowd_dist += cp12-cp11

# --------------------------------------------------------------
# Check Plan Contiguity
# check_cluster_contiguity(stateG, ip, gen, cl_graph, min_pop_id=0)
# check_district_contiguity(stateG, ip, gen, cl_graph)



# --------------------------------------------------------------
# Advance the generation
gen += 1

# --------------------------------------------------------------
# Delete previous cl_graphs and dist_graphs
for i in range(len(cl_graph[gen-1])): cl_graph[gen-1][i] = None
for i in range(len(dist_graph[gen-1])): dist_graph[gen-1][i] = None

ip.front0_sizes = stateG.front0_sizes

# --------------------------------------------------------------
# Times
print(f"time for initialize_graph = {cp0-start:.2f}")
print(f"time for create_attr_graph = {cp1-cp0:.2f}")
print(f"time for shortest_paths = {cp2-cp1:.2f}")
print(f"time for shortest path lengths = {cp3-cp2:.2f}")
print(f"time for initialize_clustering = {cp4-cp3:.2f}")
print(f"time for create cluster graphs = {cp5-cp4:.2f}")
print(f"time for create_cut_line = {time_cut_line:.2f}")
print(f"time for crossover = {time_crossover:.2f}")
print(f"time for mutation = {time_mutation:.2f}")
print(f"time for create children cluster graphs = {time_cluster_g:.2f}")
print(f"time for generate_plans = {time_gen_plans:.2f}")
print(f"time for nondominated_sorting = {time_nondom_sort:.2f}")
print(f"time for crowding_distance = {time_crowd_dist:.2f}")

end = time.time()
ip.runtime = end-start

#------------------------------------------------------------
# df0: Starting data
input_data = [list(ip.__dict__.values())]
input_columns = list(ip.__dict__.keys())
df0 = pd.DataFrame(input_data, columns=input_columns)


#------------------------------------------------------------
# df1: Assigning Precincts
cols1 = ['Generation', 'pop_id', 'node', 'cluster', 'district']
```

```
non = stateG.number_of_nodes()
ps = ip.pop_size
ng = ip.num_generations
f0s = stateG.front0_sizes[-1]
if ng <= 4:
    range_rows = range(non*ps*2*ng + non*f0s)
    range_gen = range(ng+1)
    range_ps = range(ps*2)
    num_to_del = -(2*ps-f0s)*non
else: #If the number of generations is too large, only record last front
    range_rows = range(non*ps*2*ng,
        non*ps*2*ng+min(front_lengths[0],ps)*non)
    range_gen = range(ng, ng+1)
    range_ps = range(min(front_lengths[0],ps))
    num_to_del = None
gen_col = [math.floor(i/(non*ps*2)) for i in range_rows]
id_col = [math.floor(i/non) % (2*ps) for i in range_rows]
node_col = [(i%non)+1 for i in range_rows]

cluster_col = [stateG.nodes[n]['gen'][gen][pop_id][0] for gen in
    range_gen for pop_id in range_ps for n in stateG.nodes]
district_col = [stateG.nodes[n]['gen'][gen][pop_id][1] for gen in
    range_gen for pop_id in range_ps for n in stateG.nodes]

cluster_col = cluster_col[:num_to_del]
district_col = district_col[:num_to_del]

# ----------------------------------------------------------------
# df2: For recording the metrics of each plan
cols2 = ['Generation', 'pop_id', 'rank'] + [ip.long_met_names[met] for
    met in ip.metrics if ip.metrics[met] == True ]
num_rows2 = (ng)*ps*2 + stateG.front0_sizes[-1]
gen_col2 = [math.floor(i/(2*ps)) for i in range(num_rows2)]
id_col2 = [i % (2*ps) for i in range(num_rows2)]
for idx in range(stateG.front0_sizes[-1]):
    front_col2[gen*ps*2 + idx] = 0 #Assigns the last generation rank 0
        (because they necessarily must be)


if ip.metrics['dpop']: pop_col = [stateG.pop_dev[gen][pop_id] for gen
    in range(ng+1) for pop_id in range(ps*2)]
if ip.metrics['comp']: comp_col = [stateG.comp[gen][pop_id] for gen in
    range(ng+1) for pop_id in range(ps*2)]
if ip.metrics['eg']: eg_col = [stateG.eg[gen][pop_id] for gen in
    range(ng+1) for pop_id in range(ps*2)]
if ip.metrics['mm']: mm_col = [stateG.mm[gen][pop_id] for gen in
    range(ng+1) for pop_id in range(ps*2)]
if ip.metrics['cs']: cs_col = [stateG.cs[gen][pop_id] for gen in
    range(ng+1) for pop_id in range(ps*2)]
if ip.metrics['egu']: egu_col = [stateG.egu[gen][pop_id] for gen in
    range(ng+1) for pop_id in range(ps*2)]

#num_to_del2 = 2*ps - max(stateG.front0_sizes[-1], ps)
num_to_del2 = 2*ps - stateG.front0_sizes[-1]
if ip.metrics['dpop']: pop_col = pop_col[:-num_to_del2]
if ip.metrics['comp']: comp_col = comp_col[:-num_to_del2]
if ip.metrics['eg']: eg_col = eg_col[:-num_to_del2]
if ip.metrics['mm']: mm_col = mm_col[:-num_to_del2]
if ip.metrics['cs']: cs_col = cs_col[:-num_to_del2]
if ip.metrics['egu']: egu_col = egu_col[:-num_to_del2]
front_col2 = front_col2[:-num_to_del2]

metlist = []
if ip.metrics['dpop']: metlist.append([x/ip.ub["dpop"] for x in
    pop_col])
if ip.metrics['comp']: metlist.append([x/ip.ub["comp"] for x in
    comp_col])
if ip.metrics['eg']: metlist.append([x/ip.ub["eg"] for x in eg_col])
if ip.metrics['mm']: metlist.append([x/ip.ub["mm"] for x in mm_col])
if ip.metrics['cs']: metlist.append([x/ip.ub["cs"] for x in cs_col])
if ip.metrics['egu']: metlist.append([x/ip.ub["egu"] for x in egu_col])

hvolume = hypervolume(ip, gen_col2, front_col2, metlist)

# num_to_del3 = -(ps - min(stateG.front0_sizes[-1], ps))
# if num_to_del3 == 0: num_to_del3 = None
# gen_col2 = gen_col2[:num_to_del3]
# id_col2 = id_col2[:num_to_del3]

df1_dict = {} #Contains GU assignments
df1_dict['Generation'] = gen_col
df1_dict['pop_id'] = id_col
df1_dict['node'] = node_col
df1_dict['cluster'] = cluster_col
df1_dict['district'] = district_col

df2_dict = {} #Contains metrics
df2_dict['Generation'] = gen_col2
df2_dict['pop_id'] = id_col2
df2_dict['rank'] = front_col2
if ip.metrics['dpop']: df2_dict['Pop Dev'] = pop_col
if ip.metrics['comp']: df2_dict['Compactness'] = comp_col
if ip.metrics['eg']: df2_dict['Eff Gap'] = eg_col
if ip.metrics['mm']: df2_dict['Med Mean'] = mm_col
if ip.metrics['cs']: df2_dict['County Splits'] = cs_col
if ip.metrics['egu']: df2_dict['Excess GUs'] = egu_col

# ------------------------------------------------------------
# df3: Generational data
hvstring = [met for met in list(ip.metrics.keys()) if ip.metrics[met]]
hvstring.insert(0, 'hypervolume')
hvstring = ' '.join(hvstring)
cols3 = [hvstring]
df3_dict = {} #Contains generational measures
df3_dict[hvstring] = hvolume

df1 = pd.DataFrame(data=df1_dict, columns=cols1, dtype=np.int16)
df2 = pd.DataFrame(data=df2_dict, columns=cols2)
df3 = pd.DataFrame(data=df3_dict, columns=cols3)
```

```
    met_string = ''
    for met in ip.metrics:
        if ip.metrics[met]==True: met_string += 'O'
        else: met_string += 'X'

    excel_doc_name = "NSGA2_SC_" + met_string + timetxt + ".xlsx"
    excel_writer = pd.ExcelWriter(excel_doc_name, engine='xlsxwriter')

    df0.to_excel(excel_writer, sheet_name='Input Data')
    df1.to_excel(excel_writer, sheet_name='GU Assignment')
    df2.to_excel(excel_writer, sheet_name='Metrics')
    df3.to_excel(excel_writer, sheet_name='Generational Data')

    excel_writer.save()

    #---------------------------------------------------------------------
    #Graphing
    last_front = [None] * min(front_lengths[0], ps)
    last_gen = df2[df2.Generation == ip.num_generations]

    try:
        with open('county_boundaries.pkl', 'rb') as fp: county_boundaries =
            pickle.load(fp)
    except FileNotFoundError:
        county_boundaries = Graph.from_file(
            ip.county_file,
            adjacency="rook",
            reproject="True",
            ignore_errors="True"
        )
        with open('county_boundaries.pkl', 'wb') as fp:
            pickle.dump(county_boundaries, fp)

    for i in range(len(last_front)):
        last_front[i] = df1[(df1['Generation'] == ng) & (df1['pop_id'] ==
            i)][['node','district']]
        stateG.data = stateG.data.merge(right=last_front[i],
            left_on='OBJECTID', right_on='node', validate='1:1')
        stateG.data = stateG.data.drop('node', axis=1)
        stateG.data = stateG.data.rename(columns={'district': f'plan{i}'})

    if len(fronts[0][0].metrics) == 2:
        plot_front(fronts, ip)

    plot_plans(stateG, county_boundaries)
    plot_hvolume(hvolume)

    print("finished")

if __name__ == "__main__":
    rseed = secrets.randbelow(sys.maxsize)
    #rseed = 556334154206732606 4
    random.seed(rseed)
    print("random seed is", rseed)
    if len(sys.argv) == 1:
        main('NSGA2_input3.txt', rseed) #Use file name if reading from file
    elif len(sys.argv) == 2:
        input_file = sys.argv[1]
        main(input_file, rseed)
```

Appendix E VNSGA-II Codes

```
    #Vanneschi's Algorithm, as implemented by Blake Splitter
import secrets
import random
import sys
import pickle
from gerrychain import Graph
import numpy as np
import networkx as nx
import math
import datetime
import time
import matplotlib.pyplot as plt
from pymoo.indicators.hv import HV
import pandas as pd
import copy
import matplotlib.colors as mcolors

class input_c: #short for 'input class'
    def __init__(self, json_file, ps, mp, nd, ng, met, cf, lmn, rnd, rad):
        self.json_file = json_file #Input file for the graph
        self.pop_size = ps #Population size
        self.mutation_probability = mp #Mutaiton probability
        self.num_districts = nd #Number of districts
        self.num_generations = ng #Number of generations
        self.metrics = met #Metrics dictionary. Contains either True or
           False for each metric
        self.county_file = cf #File countaining counties
        self.long_met_names = lmn #Long metric names
        self.randomness = rnd #Randomness level
        self.radius = rad #Radius for contiguity checks. Higher values are
           more thorough.

    def __repr__(self):
        met_string = ''
        for met in self.metrics:
            if self.metrics[met]==True: met_string += 'O'
            else: met_string += 'X'
        return f"<INPUT: pop_size = {self.pop_size}, num_gens =
           {self.num_generations}, mets = [{met_string}]>"

    @classmethod
    def Load_from_file(cls, file_name):
        """Reads the input from a file"""
        with open(file_name) as f:
            lines = f.readlines()
        json_file = lines[0].strip()
        pop_size = int(lines[1])
        mut_prob = float(lines[2])
        num_districts = int(lines[3])
        num_generations = int(lines[4])
        metrics = {"dpop": eval(lines[5].strip().lower().capitalize()),
                   "comp": eval(lines[6].strip().lower().capitalize()),
                   "eg": eval(lines[7].strip().lower().capitalize()),
                   "mm": eval(lines[8].strip().lower().capitalize()),
                   "cs": eval(lines[9].strip().lower().capitalize()),
                   "egu": eval(lines[10].strip().lower().capitalize())
                   }
        county_file = lines[11][0:-1]
        met_long_names = {"dpop": lines[12].strip(),
                          "comp": lines[13].strip(),
                          "eg": lines[14].strip(),
                          "mm": lines[15].strip(),
                          "cs": lines[16].strip(),
                          "egu": lines[17].strip(),
                         }
        randomness = int(lines[18])
        radius = float(lines[19])
        return cls(
            json_file=json_file,
            ps=pop_size,
            mp=mut_prob,
            nd=num_districts,
            ng=num_generations,
            met=metrics,
            cf=county_file,
            lmn=met_long_names,
            rnd=randomness,
            rad=radius
        )


def total_pop(G, ip):
    """Calculates the total population of the graph"""
    try:
        return ip.total_pop
    except AttributeError:
        ip.total_pop = sum(dict(G.nodes("POPULATION")).values())
        return ip.total_pop


def target_pop(G, ip):
    """Calculates the target population for each district"""
    try:
        return ip.ideal_pop
    except AttributeError:  # In the event that ideal_pop has not yet been
       calculated
        ip.total_pop = sum(dict(G.nodes("POPULATION")).values())
        ip.ideal_pop = ip.total_pop/ip.num_districts
        return ip.ideal_pop

def make_nbr_graphs(G, radius=3):
    """Makes neighbor graphs for every node at a distance of 'radius' away.
    G: Base graph.
    radius : Include all neighbors of distance<=radius from n. Default 3"""
    if radius == float('inf'): return None #No point in doing this
       calculation if radius = infinity.
    nbr_graphs = {}
    for n in G:
        nbr_graphs[n] = nx.ego_graph(G, n, radius=radius) #All
            neighborhoods of radius 3 for each node
        #print(f"created nbr_graphs[{n}]")
    return nbr_graphs

def initialize_graph(ip):
    """Builds a graph based on a json file supplied by the user"""
    try:
        with open('stateG.pkl', 'rb') as fp: G = pickle.load(fp)
    except FileNotFoundError:
        cols_to_add = ['OBJECTID', 'ID', 'VTD', 'COUNTY', 'STATE', 'NAME',
            'POPULATION',
                    'CountyName', 'Reg_Voters', 'PresBlue', 'PresRed',
                        'PresOther',
                    'SenBlue', 'SenRed', 'SenOther', 'Cong2011', 'Sen2011',
                        'House2011',
                    'Cong2021', 'Sen2021', 'House2021', 'geometry']
        try:
            G = Graph.from_file(
                ip.json_file,
                adjacency="rook",
                cols_to_add=cols_to_add,
                reproject="True",
                ignore_errors="True"
            )
        except KeyError:
            G = Graph.from_file(
                ip.json_file,
                adjacency="rook",
                cols_to_add=None,  # Will add all columns
                reproject="True",
                ignore_errors="True"
            )

        #Initialize front[0] sizes
        G.front0_sizes = []

        #Save stateG if it hasn't been saved before
        with open('stateG.pkl', 'wb') as fp: pickle.dump(G, fp)

    #Initialize the metric keepers
    if ip.metrics['dpop'] == True:
        G.pop_dev = np.zeros((ip.num_generations+1, 2*ip.pop_size))
    if ip.metrics['comp'] == True:
        G.comp = np.zeros((ip.num_generations+1, 2*ip.pop_size))
    if ip.metrics['eg'] == True:
        G.eg = np.zeros((ip.num_generations+1, 2*ip.pop_size))
    if ip.metrics['mm'] == True:
        G.mm = np.zeros((ip.num_generations+1, 2*ip.pop_size))
    if ip.metrics['cs'] == True:
        G.cs = np.zeros((ip.num_generations+1, 2*ip.pop_size))
    if ip.metrics['egu'] == True:
        G.egu = np.zeros((ip.num_generations+1, 2*ip.pop_size))
    G.metrics = np.zeros((ip.num_generations+1, 2*ip.pop_size,
        sum(ip.metrics.values())))
    ip.num_GUs = G.number_of_nodes()
    county_list = set()
    for n in G.nodes:
        G.nodes[n]['district'] = np.full(shape=(ip.num_generations+1,
            ip.pop_size*2), fill_value=-1)
        # 1 represents that each entry of G.nodes[n]['gen'][g][pop_id] will
            return a district value
        county_list.add(G.nodes[n]['COUNTY'])
    county_list = sorted(county_list)
    # Long county name to short county name. e.g. longc2shortc['45003'] = 1
    longc2shortc = {}
    for idx, c in enumerate(county_list):
        longc2shortc[c] = idx
    for n in G.nodes:
        G.nodes[n]['county_num'] = longc2shortc[G.nodes[n]['COUNTY']]
        if G.nodes[n]['boundary_perim'] < 0: G.nodes[n]['bounary_perim'] = 0
        #print(f"G.nodes[{n}]['boundary_perim'] =
            {G.nodes[n]['boundary_perim']}")
    return G

def create_attr_graph(G, ip, attr=None, gen_id=None):
    """Creates a graph that merges all nodes of a certain attribute (attr)
        together
    'G' is NOT amended in this function
    G: Graph
    ip: input
    attr: (optional, can't be entered with 'gen_id') Attribute to group by
    gen_id: (optional, can't be entered with 'attr') If grouping a
        districting plan, use the tuple (generation, districting pop_id)"""

    if attr == None and gen_id == None:
        raise ValueError("Either 'attr' or 'gen_id' must be defined")
    if attr != None and gen_id != None:
        raise ValueError("Either 'attr' or 'gen_id' must be defined, but
            not both")

    gen = None
    pop_id = None
    if gen_id != None:
        gen = gen_id[0]
        pop_id = gen_id[1]

    cg = nx.Graph() #cg = contracted graph
    node2supernode = {}
    node_attr_list = [
        'area',
        'POPULATION',
        'Reg_Voters',
        'PresBlue',
        'PresRed',
        'PresOther',
        'SenBlue',
        'SenRed',
        'SenOther',
        ]
    # Removes all edges from graph G where one end has different attribute
        (attr) values than the other end
    #boundary_edges = []
    for e in G.edges():
        n0 = e[0]
        n1 = e[1]

        if attr != None: # we are separating by attribute
            sn0 = G.nodes[n0][attr]
            sn1 = G.nodes[n1][attr]
        elif gen_id != None: #we are separating by districting
            sn0 = G.nodes[n0]['district'][gen][pop_id]
            sn1 = G.nodes[n1]['district'][gen][pop_id]
        else: raise ValueError("Either attr or gen_id should be defined")
        node2supernode[n0] = sn0
        node2supernode[n1] = sn1

        if sn0 != sn1: #i.e, this edge represents a boundary for the
           attribute
            #boundary_edges += [e]

            # try: supernode2node[sn0].append(n0)
            # except KeyError: supernode2node[sn0] = [n0]
            # try: supernode2node[sn1].append(n1)
            # except KeyError: supernode2node[sn1] = [n1]
            cg.add_node(sn0)
            cg.add_node(sn1)
            cg.add_edge(sn0, sn1)

            #Adds shared perimeter to total perimeter attribute for nodes
            try: cg.nodes[sn0]['total_perim'] += G.edges[(n0,
                n1)]['shared_perim']
            except KeyError: cg.nodes[sn0]['total_perim'] = G.edges[(n0,
                n1)]['shared_perim']
            try: cg.nodes[sn1]['total_perim'] += G.edges[(n0,
                n1)]['shared_perim']
            except KeyError: cg.nodes[sn1]['total_perim'] = G.edges[(n0,
                n1)]['shared_perim']

            #Add edge attributes to cg *only* if it was a boundary edge
            try: cg[sn0][sn1]['shared_perim'] += G.edges[(n0,
                n1)]['shared_perim']
            except KeyError: cg.edges[(sn0, sn1)]['shared_perim'] =
                G.edges[(n0, n1)]['shared_perim']

            # Creates a list of all edges that created this supernode edge
            try: cg[sn0][sn1]['contributing_edges'] += [e]
            except KeyError: cg[sn0][sn1]['contributing_edges'] = [e]


    for sn in cg.nodes():
        cg.nodes[sn]['boundary_perim'] = 0 ###Maybe a more elegant fix
           later?
```

```
        cg.nodes[sn]['GUs'] = []



    for n in G.nodes():
        sn = node2supernode[n]
        cg.nodes[sn]['GUs'].append(n)


        #Initialize and populate the attributes
        for nattr in node_attr_list:
            if nattr in cg.nodes[sn]: cg.nodes[sn][nattr] +=
                G.nodes[n][nattr]
            else: cg.nodes[sn][nattr] = G.nodes[n][nattr]

        #For boundary_perim
        try:
            cg.nodes[sn]['boundary_perim'] += G.nodes[n]['boundary_perim']
            cg.nodes[sn]['total_perim'] += G.nodes[n]['boundary_perim']
        except KeyError: #If supernode isn't on a state boundary
            cg.nodes[sn]['boundary_perim'] += 0
            cg.nodes[sn]['total_perim'] += 0

        # Names the county if the attribute is 'COUNTY' (done more times
           than necessary, but that's ok)
        if attr == 'COUNTY' or attr == 'county_num':
            cg.nodes[sn]['CountyName'] = G.nodes[n]['CountyName']

    # If we are creating the district plan, calculate the metrics
    if gen_id != None:
        cg.total_pop = total_pop(G, ip)
        cg.ideal_pop = target_pop(G, ip)
        cg.CDI_matrix = np.zeros((ip.num_districts, ip.num_counties),
           dtype=np.int16)

        for d in cg.nodes: #'d' represents a district
            cg.nodes[d]['pp_compactness'] = 4 * math.pi *
                cg.nodes[d]['area'] / (cg.nodes[d]['total_perim'] ** 2)
            cg.nodes[d]['shifted_pp_compactness'] = 1 -
                cg.nodes[d]['pp_compactness']
            cg.nodes[d]['total_Pvotes'] = cg.nodes[d]['PresBlue'] +
                cg.nodes[d]['PresRed'] + cg.nodes[d]['PresOther']
            cg.nodes[d]['total_Svotes'] = cg.nodes[d]['SenBlue'] +
                cg.nodes[d]['SenRed'] + cg.nodes[d]['SenOther']
            cg.nodes[d]['blue_Pshare'] = cg.nodes[d]['PresBlue'] /
                cg.nodes[d]['total_Pvotes'] if
                cg.nodes[d]['total_Pvotes']>0 else 0
            cg.nodes[d]['red_Pshare'] = cg.nodes[d]['PresRed'] /
                cg.nodes[d]['total_Pvotes'] if
                cg.nodes[d]['total_Pvotes']>0 else 0
            cg.nodes[d]['other_Pshare'] = cg.nodes[d]['PresOther'] /
                cg.nodes[d]['total_Pvotes'] if
                cg.nodes[d]['total_Pvotes']>0 else 0
            cg.nodes[d]['blue_Sshare'] = cg.nodes[d]['SenBlue'] /
                cg.nodes[d]['total_Svotes'] if
                cg.nodes[d]['total_Svotes']>0 else 0
            cg.nodes[d]['red_Sshare'] = cg.nodes[d]['SenRed'] /
                cg.nodes[d]['total_Svotes'] if
                cg.nodes[d]['total_Svotes']>0 else 0
            cg.nodes[d]['other_Sshare'] = cg.nodes[d]['SenOther'] /
                cg.nodes[d]['total_Svotes'] if
                cg.nodes[d]['total_Svotes']>0 else 0
            cg.nodes[d]['win_threshold_P'] =
                math.ceil(cg.nodes[d]['total_Pvotes']/2 + 0.5)
            cg.nodes[d]['win_threshold_S'] =
                math.ceil(cg.nodes[d]['total_Svotes']/2 + 0.5)
            cg.nodes[d]['Rwasted_votes_P'] = cg.nodes[d]['PresRed'] -
                cg.nodes[d]['win_threshold_P'] if cg.nodes[d]['PresRed'] >
                cg.nodes[d]['PresBlue'] else cg.nodes[d]['PresRed']
            cg.nodes[d]['Bwasted_votes_P'] = cg.nodes[d]['PresBlue'] -
                cg.nodes[d]['win_threshold_P'] if cg.nodes[d]['PresRed'] <=
                cg.nodes[d]['PresBlue'] else cg.nodes[d]['PresBlue']
            for gu in cg.nodes[d]['GUs']:
                county = G.nodes[gu]['county_num']
                cg.CDI_matrix[d][county] += 1

        cg.pop_dev_sum = round(sum(abs(p-cg.ideal_pop) for p in
           dict(cg.nodes("POPULATION")).values()))
        cg.pop_dev_max = round(max(abs(p-cg.ideal_pop) for p in
           dict(cg.nodes("POPULATION")).values()))
        cg.exc_county_splits = np.count_nonzero(cg.CDI_matrix) -
           max(ip.num_counties, ip.num_districts)
        cg.exc_gus = sum(np.sum(cg.CDI_matrix, axis=0) -
           np.max(cg.CDI_matrix, axis=0))
        cg.cs = cg.exc_county_splits
        cg.egu = cg.exc_gus
        cg.pp_comp = round(sum([cg.nodes[d]['pp_compactness'] for d in
           cg.nodes]) / cg.number_of_nodes(), 3)
        cg.shifted_pp_comp = 1-cg.pp_comp
        bsP_list = [cg.nodes[d]['blue_Pshare'] for d in cg.nodes]  # bsP =
           blue share for president
        bsP_list = sorted(bsP_list)
        if ip.num_districts % 2 == 0:  # even
            bsP_median = (bsP_list[int(ip.num_districts/2)] +
                bsP_list[ip.num_districts/2-1])/2
        else:  # odd
            bsP_median = bsP_list[int(ip.num_districts/2-0.5)]
        bsP_mean = sum(bsP_list)/ip.num_districts
        cg.mm = abs(bsP_median - bsP_mean)  # median-mean
        bwv_P = sum([cg.nodes[d]['Bwasted_votes_P'] for d in cg.nodes])
        rwv_P = sum([cg.nodes[d]['Rwasted_votes_P'] for d in cg.nodes])
        cg.total_Pvotes = sum([cg.nodes[d]['total_Pvotes'] for d in
           cg.nodes])
        cg.eg = abs((bwv_P - rwv_P)/cg.total_Pvotes)  # absolute efficiency
           gap
        cg.all_metrics = [cg.pop_dev_sum, cg.shifted_pp_comp, cg.eg, cg.mm,
           cg.cs, cg.egu]
        cg.metric_keys = ['dpop', 'comp', 'eg', 'mm', 'cs', 'egu']
        cg.metrics = [m for idx, m in enumerate(cg.all_metrics) if
           ip.metrics[cg.metric_keys[idx]] == True]
        #Place these metrics in G as well
        if ip.metrics['dpop']: G.pop_dev[gen][pop_id] = cg.pop_dev_sum
        if ip.metrics['comp']: G.comp[gen][pop_id] = cg.shifted_pp_comp
        if ip.metrics['eg']: G.eg[gen][pop_id] = cg.eg
        if ip.metrics['mm']: G.mm[gen][pop_id] = cg.mm
        if ip.metrics['cs']: G.cs[gen][pop_id] = cg.cs
        if ip.metrics['egu']: G.egu[gen][pop_id] = cg.egu
        G.metrics[gen][pop_id] = cg.metrics
    return cg

def generate_plan(rng, G, ip, gen, pop_id, randomness=1):
    """Generates a districting plan based on the
    G: base graph
    ip: input
    gen: current generation
    pop_id: the population ID
    randomness: (optional; default 1) the randomness level. 0:
       deterministic, 1: semi-random, 2: fully random"""
    failed_plan = True
    while failed_plan == True: #Perpertual loop that will occur until a
       viable plan is created
        used_centers, nbrlengths = generate_centers(rng, G, ip, gen,
            pop_id, randomness)
        dist2node = {} # A dictionary that details which nodes are in a
            district (dist2node[dist] = [list of nodes])
        for d in range(ip.num_districts): dist2node[d] = []
        maxsize = {} # A dictionary containing the maximum size (in nodes)
            of each district
        currsize = {} # A dictionary containing the current size (in nodes)
            of each district

        unassigned_nodes = list(G.nodes)
        viable_dists = list(range(ip.num_districts))
        distpops = [0]*ip.num_districts
        for dist_num, center in enumerate(used_centers):
            maxsize[dist_num] = float('inf') # The districts should not be
                constrained by number of nodes
            currsize[dist_num] = 1
            distpops[dist_num] += G.nodes[center]['POPULATION']
            #dist_nbrs[dist_num] = set(G.neighbors(center))
            unassigned_nodes.remove(center)
            dist2node[dist_num].append(center)
            G.nodes[center]['district'][gen][pop_id] = dist_num
            # GUs_in_center = G.nodes[center]['GUs']
            # for gu in GUs_in_center:
            #     # Assigns a distict number to the GU in the underlying
                graph
            #     G.nodes[gu]['gen'][gen][pop_id][1] = dist_num
            # cl_graph[gen][pop_id].nodes[center]['districting'] = dist_num

        failed_plan = False
        while unassigned_nodes != []:
            if viable_dists == []: #If no districts can be grown and/or
                they are over 2*ideal_pop
                failed_plan = True
                print(f"Gen {gen}. Failed districting {pop_id} (no viable
                    districts).")
                clear_plan(G, gen, pop_id)
                break #Breaks from 'while unassigned_nodes != []' loop;
                    Restarts the 'while failed_plan==False' loop

            if randomness==0: #deterministic district choices. Picks least
                populated viable district
                dist = -1
                distpops_copy = distpops.copy()
                while dist not in viable_dists:
                    if dist != -1:
                        distpops_copy[dist] = np.inf
                    dist = distpops_copy.index(min(distpops_copy))
            elif randomness==1: #pseudo-random district choices
                pop_weights = [total_pop(G, ip)-distpops[i] for i in
                    range(len(viable_dists))]
                mdw = min(pop_weights)
                pop_weights = [pop_weights[i] - mdw+1 for i in
                    range(len(pop_weights))]
                dist = rng.choices(viable_dists, weights=pop_weights,
                    k=1)[0]
            elif randomness==2: #random district choices
                dist = rng.choice(viable_dists)
            if nbrlengths[dist] == {}: #Can occur if previous removal of GU
                from another district impacts this district
                viable_dists.remove(dist)
                continue

            currsize, maxsize, unassigned_nodes, dist2node, added_node,
                nbrlengths = grow_district(rng, G, dist, gen, pop_id,
                currsize, maxsize, unassigned_nodes, dist2node, nbrlengths)
            if currsize[dist] == maxsize[dist]:
                # This occurs if dist cannot add any more GUs because all
                    surrounding GUs are assigned
                viable_dists.remove(dist)
            if added_node != None:
                distpops[dist] += G.nodes[added_node]['POPULATION']
                # If a district has exceeded twice the ideal population,
                    stop adding GUs to it
                if distpops[dist] > target_pop(G,ip)*2:
                    try: viable_dists.remove(dist)
                    except ValueError: pass #May have already removed dist
                        from viable_dists in currsize == maxsize check


        if any(distpops[i] < 0.15*target_pop(G, ip) for i in
            range(ip.num_districts)) and failed_plan == False: #'and'
            present so we clear plan only once
            failed_plan = True
            clear_plan(G, gen, pop_id)
            print(f"Gen {gen}. Failed districting {pop_id} (underpopulated
                district).")
        if failed_plan == False:
            print(f"Gen {gen}. Created plan {pop_id}.")
            break  # breaks from 'while True' loop because a valid plan was
                produced


def generate_centers(rng, G, ip, gen, pop_id, randomness=1,
    num_sets_centers=5):
    """Determines which GUs we should center the districts around
    G: graph of GUs
    randomness: the randomness level for choosing the centers
        (0=deterministic; 1=semi-random; 2=completely random)
    num_sets_centers: The number of sets of centers that will be chosen if
        randomness level 1 is used
    """
    if randomness == 2: #completely random centers
        used_centers = rng.sample(list(G.nodes),  ip.num_districts)
    elif randomness == 1: #centers chosen that are far apart
        centers = [None]*num_sets_centers
        summed_center_length = [0 for _ in range(num_sets_centers)]
        for j in range(num_sets_centers): #Randomly selects n GUs which
            will form the base of our districts
            centers[j] = rng.sample(list(G.nodes), ip.num_districts)
            for c_idx, c in enumerate(centers[j]):
                for d_idx, d in enumerate(centers[j]):
                    #We find the summed graph distance between all centers
                    if c_idx < d_idx: summed_center_length[j] +=
                        nx.shortest_path_length(G, source=c, target=d)
        used_centers_idx =
            summed_center_length.index(max(summed_center_length)) #Finds
            the index for the centers set with most dispersion
        used_centers = centers[used_centers_idx]

    elif randomness == 0: #deterministic centers chosen by highest
        population that are far enough apart
        pop_dict = dict(G.nodes("POPULATION"))
        sorted_pop_dict = dict(sorted(pop_dict.items(), key=lambda item:
            item[1], reverse=True))
        cl_list = list(sorted_pop_dict.keys())
        used_centers = []
        aspl = nx.average_shortest_path_length(G)
        min_distance = aspl*0.8 #minimum mutual distance that the nodes
            must be separated by
        k=0
        while len(used_centers) < ip.num_districts: #Will select the
            [ip.num_districts] centers that are highest populated *and* at
            least a certain distance apart from each other
            if len(used_centers) == 0: used_centers.append(cl_list[k])
            elif k<len(cl_list)-1: #If the candidate center is sufficiently
                far from all current centers, add it to the list
                k+=1
                #if min([nx.shortest_path_length(cl_graph[gen][pop_id],
                    source=cl_list[k], target=cent) for cent in
                    used_centers]) > diameter/math.sqrt(ip.num_districts):
```

```
                if min([nx.shortest_path_length(G, source=cl_list[k],
                    target=cent) for cent in used_centers]) > min_distance:
                     used_centers.append(cl_list[k])
            elif k == len(cl_list)-1: #We have explored all GUs and cannot
                find centers that are sufficiently far apart
                min_distance = min_distance*0.8
                print(f"reducing minimum distance requirement. New min dist
                    = {min_distance}")
                used_centers = []
                k=0

    #Starts a dictionary that records nbrlengths for each district
    nbrlengths = {} #nbrlengths[dist][nbr] = length_with_dist
    for dist, c in enumerate(used_centers):
        nbrlengths[dist] = {}
        for nbr in G.neighbors(c):
            if G.nodes[nbr]['district'][gen][pop_id] == -1:
                nbrlengths[dist][nbr] = G.edges[(c,nbr)]['shared_perim']
        if randomness <= 1: nbrlengths[dist] =
            dict(sorted(nbrlengths[dist].items(), key=lambda x:x[1],
            reverse=True))
    return used_centers, nbrlengths

def grow_district(rng, G, dist, gen, pop_id, currsize, maxsize,
    unassigned_nodes, dist2node, nbrlengths):
    """Grows the group district that 'node' is in by one GU unless it can't
    G: Graph
    dist: the district to grow
    gen: The generation we're on
    pop_id: The pop_id of the plan under consideration
    currsize: a dictionary containing the current size (in nodes) of each
        group
    maxsize: a dictionary containing the the maximum size (in nodes) of
        each group
    unassigned_nodes: A list of nodes that have not yet been assigned a
        group
    dist2node: A dictionary detailing all the nodes in each district.
        dist2node[dist] = [list of nodes]
    nbrlengths: A dictionary detailing the boundary length with the
        district. nbrlengths[dist][node] = length"""

    if dist == -1: raise ValueError("dist should be nonnegative")

    nbrs = list(nbrlengths[dist].keys())
    nbrs = weighted_shuffle(rng, nbrs,
        weights=list(nbrlengths[dist].values())) #neighboring nodes for dist

    i=-1 #for cycling for nbrs indices
    while True: # We break once a nbr is found or all nbrs are exhausted
        if len(nbrs) == 0: break
        i += 1
        nbr = nbrs[i]
        nbr_distnum = G.nodes[nbr]['district'][gen][pop_id]
        if nbr_distnum == -1:  # If the nbr has not been assigned a
            district already
            G.nodes[nbr]['district'][gen][pop_id] = dist #assigns nbr to
                dist

            dist2node[dist] += [nbr]
            unassigned_nodes.remove(nbr)
            currsize[dist] += 1
            nbrlengths = update_nbrlengths(G, nbrlengths, gen, pop_id, nbr,
                dist)

            if nbrlengths != None and nbrlengths[dist] == {}: maxsize[dist]
                = currsize[dist]
            return currsize, maxsize, unassigned_nodes, dist2node, nbr,
                nbrlengths
        else: # If the nbr has been assigned
            if i == len(nbrs)-1:
                break
    # If we made it this far, it's because we checked all neighbors and
        can't append any to the current group
    maxsize[dist] = currsize[dist] # Adjusts the maxsize of this group so
        we don't iterate through its neighbors again
    return currsize, maxsize, unassigned_nodes, dist2node, None, nbrlengths

def weighted_shuffle(rng, items, weights, rvs=True):
    """Given a list of items and weights for each one, we return a list of
        those shuffled items where higher weights are more probable"""
    order = sorted(range(len(items)), key=lambda i: rng.random() ** (1.0 /
        weights[i]), reverse=rvs)
    return [items[i] for i in order]

def update_nbrlengths(G, nbrlengths, gen, pop_id, newnbr, dist):
    """Updates the nbrlengths dictionary after newnbr was added to
        district"""

    del nbrlengths[dist][newnbr] #Since newnbr is in dist now, dist doesn't
        neighbor newnbr
    for nn in G.neighbors(newnbr):
        if G.nodes[nn]['district'][gen][pop_id] == -1:
            try:
                nbrlengths[dist][nn] += G.edges[(nn,
                    newnbr)]['shared_perim']
            except KeyError:
                nbrlengths[dist][nn] = G.edges[(nn, newnbr)]['shared_perim']
        elif G.nodes[nn]['district'][gen][pop_id] != dist:
            nndist = G.nodes[nn]['district'][gen][pop_id]
            try: del nbrlengths[nndist][newnbr] #Occurs if nn is already
                assigned a district
            except KeyError: pass
    nbrlengths[dist] = dict(sorted(nbrlengths[dist].items(), key=lambda
        x:x[1], reverse=True))
    return nbrlengths

def clear_plan(G, gen, pop_id):
    """Clears the districting plan in generation gen and with id pop_id"""
    for node in G.nodes:
        G.nodes[node]['district'][gen][pop_id] = -1

def tournament_selection(rng, ip, dist_graph, gen, tourn_size):
    """Determines a parent for crossover from the list of parent candidates
       p_list
    rng: random number generator for code repeatability
    ip: input
    dist_graph: the array of district graphs. dist_graph[gen][pop_id] =
       <Graph with d nodes>
    gen: current generation
    tourn_size: number of individuals to compare.
    p_list: a list of parent candidates. Each candidate is a district
       graph"""

    f_id, fronts = nondominated_sorting(dist_graph[gen][0:ip.pop_size])



    # Establish the median for each front
    median_index = [None]*len(fronts)
    triplets = [None]*len(fronts)
    ranks = {}
    med_dist = {}
    for i, front in enumerate(fronts):
        # Create a list of tuples (graph_id, pop_dev_sum)
        graph_id_pairs = [(graph_id, dg.pop_dev_sum) for graph_id, dg in
            zip(f_id[i], fronts[i])]

        # Sort the list of tuples based on the pop_dev_sum
        sorted_graph_id_pairs = sorted(graph_id_pairs, key=lambda x: x[1])
        median_index[i] = (len(front)-1)/2
        triplets[i] = [(graph_id, pop_dev_sum, abs(index -
            median_index[i])) for index, (graph_id, pop_dev_sum) in
            enumerate(sorted_graph_id_pairs)]

    #Establish dictionary for rank and distance from median entry in its
       front
    for rank, l in enumerate(triplets):
        for trip in l:
            graph_id = trip[0]
            ranks[graph_id] = rank
            med_dist[graph_id] = trip[2]

    # Pick random parent candidates
    p_candidates_idxs = rng.sample(range(ip.pop_size), tourn_size)
    min_rank = min([ranks[p] for p in p_candidates_idxs])
    best_parents_idx = [p for p in p_candidates_idxs if ranks[p] ==
       min_rank]
    # best_parents = []
    # for i, rank in enumerate(f_id):
    #     for j, dg_idx in enumerate(rank):
    #         if dg_idx in best_parents_idx:
    #             best_parents.append(fronts[i][j])
    if len(best_parents_idx) == 1: # If one plan dominates all others
        return best_parents_idx[0]
    else:
        min_med_dist = min([med_dist[p] for p in best_parents_idx])
        for i, p in enumerate(best_parents_idx):
            if med_dist[p] == min_med_dist:
                return best_parents_idx[i]



    # p_candidates = [dist_graph[gen][idx] for idx in p_candidates_idxs]
    # p_id, p_fronts = nondominated_sorting(p_candidates)
    # if len(p_id[0]) == 1: # If one plan dominates all others
    #     return p_fronts[0][0] # Return best plan
    # else: #If front 0 has multiple plans
    #     for idx in p_id[0]:


def crossover(rng, G, ip, gen, nbr_graphs, pop_id, p1, p2, c,
    single_child=True):
    """Performs the crossover step of NSGA-II, creating two children
    rng: random number generator for code repeatability
    G: Graph
    ip: input
    gen: current generation
    nbr_graphs: the k-layer neighbor graph dictionary for all nodes.
        nbr_graphs[n] = [list of nodes k away from n]
    pop_id_idx: index of the loop
    p1: population ID for parent 1
    p2: population ID for parent 2
    c: crossover point. Should be an integer <= G.num_of_nodes
    single_child: denotes whether we should make a single child instead of
        2"""
    # if not single_child:
    #     #Copy parent to s1
    #     s1_idx = 2*pop_id_idx + ip.pop_size
    #     s2_idx = 2*pop_id_idx + ip.pop_size + 1
    #     for n in G.nodes: #copies p1 and p2 to s1 and s2
    #         try: G.nodes[n]['district'][gen][s1_idx] =
        G.nodes[n]['district'][gen][p1]
    #         except IndexError: pass
    #         try: G.nodes[n]['district'][gen][s2_idx] =
        G.nodes[n]['district'][gen][p2]
    #         except IndexError: pass
    #     for n in G.nodes:
    #         d1 = G.nodes[n]['district'][gen][p1] #n's district in p1
    #         d2 = G.nodes[n]['district'][gen][p2] #n's district in p2
    #         if d1 == -1: raise ValueError("d1 shouldn't be -1.")
    #         elif d2 == -1: raise ValueError("d2 shouldn't be -1.")
    #         n_nbrs = G.neighbors(n)
    #         if n<=c: #then see if
        stateG.nodes[n]['district'][gen][s1_idx] can become d2
    #             if d1 == d2:
    #                 continue #if districts are the same, no change can be
        made
```

```
#               n_p1_nbrs_dists = [G.nodes[nbr]['district'][gen][p1] for
    nbr in n_nbrs]
#               if d2 not in n_p1_nbrs_dists:
#                   continue #if d2 is not present in the nbrs of n in
    p1, then no swap can be made
#               #If we make it this far, then districts are not the same,
    and making the flip is plausible
#               flip_status = flip(G, ip, n, d2, gen, s1_idx, nbr_graphs,
    ip.radius)
#               #nbrhd = nx.subgraph(nbr_graphs[n], [x for x in
    nbr_graphs[n].nodes() if G.nodes[x]['district'][gen][s1_idx] == d1
    and x!=n])
#               #if not nx.is_connected(nbrhd):
#               if flip_status == False:
#                   continue #if making the flip disconnects the graph
#               #If we make it this far, then the flip should be performed
#               #G.nodes[n]['district'][gen][s1_idx] = d2

#           if n>c and s2_idx<2*ip.pop_size: #then see if
    stateG.nodes[n]['district'][gen][s2_idx] can become d1
#               if d2 == d1:
#                   continue #if districts are the same, no change can be
    made
#               n_p2_nbrs_dists = [G.nodes[nbr]['district'][gen][p2] for
    nbr in n_nbrs]
#               if d1 not in n_p2_nbrs_dists:
#                   continue #if d1 is not present in the nbrs of n in
    p2, then no swap can be made
#               #If we make it this far, then districts are not the same,
    and making the flip is plausible
#               flip_status = flip(G, ip, n, d1, gen, s2_idx, nbr_graphs,
    ip.radius)
#               #nbrhd = nx.subgraph(nbr_graphs[n], [x for x in
    nbr_graphs[n].nodes() if G.nodes[x]['district'][gen][s2_idx] == d2
    and x!=n])
#               #nbrhd = nx.subgraph(G, [x for x in G.nodes() if
    G.nodes[x]['district'][gen][s2_idx] == d2 and x!=n])
#               #if not nx.is_connected(nbrhd):
#               if flip_status == False:
#                   continue #if making the flip disconnects the graph
#               #If we make it this far, then the flip should be performed
#               #G.nodes[n]['district'][gen][s2_idx] = d1
#       if s2_idx < 2*ip.pop_size:
#           print(f"Gen {gen}. Bred child plans {s1_idx} and {s2_idx}.")
#       else:
#           print(f"Gen {gen}. Bred child plan {s1_idx}.")
if single_child==True:
    #Copy parent to s
    #sidx = pop_id_idx + ip.pop_size
    p = rng.choice([p1,p2]) #randomly choose either p1 or p2 to be the
        base parent
    for n in G.nodes: #copies either p1 or p2 to s
        G.nodes[n]['district'][gen][pop_id] =
            G.nodes[n]['district'][gen][p]
    for n in G.nodes:
            d1 = G.nodes[n]['district'][gen][p1] #n's district in p1
            d2 = G.nodes[n]['district'][gen][p2] #n's district in p2
            if d1 == -1: raise ValueError("d1 shouldn't be -1.")
            if d2 == -1: raise ValueError("d2 shouldn't be -1.")
            if n<=c and p==p1: #then see if
                stateG.nodes[n]['district'][gen][sidx] can become d2
                if d1 == d2:
                    continue #if districts are the same, no change can be
                        made
                n_nbrs_dists = [G.nodes[nbr]['district'][gen][pop_id] for
                    nbr in G.neighbors(n)]
                if d2 not in n_nbrs_dists:
                    continue #if d2 is not present in the nbrs of n in p1,
                        then no swap can be made
                #If we make it this far, then districts are not the same,
                    and making the flip is plausible
                flip_status = flip(G, ip, n, d2, gen, pop_id, nbr_graphs,
                    ip.radius)
                #nbrhd = nx.subgraph(nbr_graphs[n], [x for x in
                    nbr_graphs[n].nodes() if
                    G.nodes[x]['district'][gen][sidx] == d1 and x!=n])
                #if not nx.is_connected(nbrhd):
                if flip_status == False:
                    continue #if making the flip disconnects the graph
                #If we make it this far, then the flip should be performed
                #G.nodes[n]['district'][gen][sidx] = d2

            if n>c and p==p2: #then see if
                stateG.nodes[n]['district'][gen][sidx] can become d1
                if d2 == d1:
                    continue #if districts are the same, no change can be
                        made
                n_nbrs_dists = [G.nodes[nbr]['district'][gen][pop_id] for
                    nbr in G.neighbors(n)]
                if d1 not in n_nbrs_dists:
                    continue #if d1 is not present in the nbrs of n in p2,
                        then no swap can be made
                #If we make it this far, then districts are not the same,
                    and making the flip is plausible
                flip_status = flip(G, ip, n, d1, gen, pop_id, nbr_graphs,
                    ip.radius)
                #nbrhd = nx.subgraph(nbr_graphs[n], [x for x in
                    nbr_graphs[n].nodes() if
                    G.nodes[x]['district'][gen][sidx] == d2 and x!=n])
                #if not nx.is_connected(nbrhd):
                if flip_status == False:
                    continue #if making the flip disconnects the graph
                #If we make it this far, then the flip should be performed
                #G.nodes[n]['district'][gen][sidx] = d1
        #print(f"Gen {gen}. Bred child plan {pop_id}.")
    else: raise RuntimeError("single_child must be True; 'False' is defunct
        right now.")


def mutation(rng, G, ip, gen, pop_id, nbr_graphs):
    """Mutates the graph G based on the input mutation probability
    rng: random number generator for code repeatability
    G: Graph
    ip: input
    gen: generation
    pop_id: the population ID
    nbr_graphs: the k-layer neighbor graph dictionary for all nodes.
        nbr_graphs[n] = [list of nodes k away from n]"""
    for n in G.nodes:
        r = rng.random() #Generates a number between 0 and 1
        if r <= ip.mutation_probability: #If True, mutate
            nbrs = G.neighbors(n)
            nbr_dists = [G.nodes[nbr]['district'][gen][pop_id] for nbr in
                nbrs]
            n_dist = G.nodes[n]['district'][gen][pop_id]
            if list(set(nbr_dists)) == [n_dist]: continue #If all neighbors
                of n are in the same district, no flip can be made
            else: #If n is on a district boundary
                nbr_dist = n_dist #temp
                while nbr_dist == n_dist: nbr_dist =
                    rng.choice(list(set(nbr_dists))) #Ensures a dist
                    selection other than n_dist
                nbrhd = nx.subgraph(nbr_graphs[n], [x for x in
                    nbr_graphs[n].nodes() if
                    G.nodes[x]['district'][gen][pop_id] == n_dist and x!=n])
                if not nx.is_connected(nbrhd): continue #if making the flip
                    disconnects the graph
                #If we make it this far, then make the mutation flip
                G.nodes[n]['district'][gen][pop_id] = nbr_dist

def flip(G, ip, n, dist, gen, pop_id, nbr_graphs, radius = float('inf'),
        dist_graph=None, structure=None):
    """Performs a flip where we move GU 'n' to distrist 'dist'. This checks
        that contiguity will be maintained.
    G: Base graph
    ip: input
    n: Node to be flipped
    dist: district that n will be moved into
    gen: current generation
    pop_id: the population ID for the flip
    nbr_graphs: the k-layer neighbor graph dictionary for all nodes.
        nbr_graphs[n] = [list of nodes k away from n]
    radius: The level of contiguity to check. Can be float('inf') or any
        positive integer.
         If float('inf'), then we collect all GUs in current district and
             see if removing n creates discontiguity.
         If a positive integer 'k' is used, we only check GUs in the current
             district that are at most k away from n, and see if removing n
             creates discontiguity.
    dist_graph: array of district graphs
    structure: (optional, default None). If we want to only consider flips
        that improve population, structure=1.
         If we only want to consider flips that improve compactness,
             structure=2.
    """
    n_dist = G.nodes[n]['district'][gen][pop_id]
    if n_dist == dist: raise ValueError(f"Cannot flip node {n} to district
        {dist} because node {n} is already in district {dist}")
    if dist not in [G.nodes[nbr]['district'][gen][pop_id] for nbr in
        G.neighbors(n)]:
        raise ValueError(f"Cannot flip node {n} to district {dist} because
            district {dist} does not border node {n}")

    structure_satisfied = (structure==None) or \
                          (structure==1 and
                              pop_improves(G,dist_graph,n,dist,gen,pop_id))
                              or \
                          (structure==2 and
                              comp_improves(G,dist_graph,n,dist,gen,pop_id))
    if structure_satisfied == False:
        return False

    if radius == float('inf'): #Check full contiguity
        nbrhd = nx.subgraph(G, [x for x in G.nodes() if
            G.nodes[x]['district'][gen][pop_id] == n_dist and x!=n])
    else: #Check k-layer contiguity
        nbrhd = nx.subgraph(nbr_graphs[n], [x for x in
            nbr_graphs[n].nodes() if G.nodes[x]['district'][gen][pop_id] ==
            n_dist and x!=n])

    if set([G.nodes[nbr]['district'][gen][pop_id] for nbr in
        G.neighbors(n)]) == {dist}:
        return False #This indicates that this district would disappear

    if nx.is_connected(nbrhd) and structure_satisfied:
        G.nodes[n]['district'][gen][pop_id] = dist #perform the flip
        if dist_graph!=None: dist_graph[gen][pop_id] = upd_dg(G, ip,
            dist_graph, gen, pop_id, n, n_dist, dist)

        return True #signifies flip was successful
    else: #flip would create discontiguity
        return False #signifies flip was unsuccessful

def upd_dg(G, ip, dist_graph, gen, pop_id, n, l_dist, e_dist):
    """Updates the dist_graph[gen][pop_id] metrics.
    G: underlying GU graph
    ip: input
    dist_graph: array of district graphs. dist_graph[gen][pop_id] = <Graph
        with d nodes>
    gen: generation
    pop_id: population ID
    n: node that was moved
    l_dist: the district that the node n is leaving
    e_dist: the district that the node n is entering"""

    cg = dist_graph[gen][pop_id]
    d_list = [l_dist, e_dist]

    cg.nodes[e_dist]['GUs'].append(n)
    cg.nodes[l_dist]['GUs'].remove(n)
```

```
for nbr in G.neighbors(n):  #Cycles through neighboring nodes to adjust
    boundary status, perimeter, and district neighbors
    nbr_dist = G.nodes[nbr]['district'][gen][pop_id]

    if nbr_dist == e_dist:
        #G[n][nbr]["Dist_Boundary"] = 0
        cg.nodes[e_dist]["total_perim"] -= G[n][nbr]["shared_perim"]
            #This is now 'interior' perimeter
        cg.nodes[l_dist]["total_perim"] -= G[n][nbr]["shared_perim"]
            #Neither n nor nbr is in l_dist
        if (l_dist, nbr_dist) in cg.edges:
            try:
                cg[l_dist][nbr_dist]['contributing_edges'].remove((n,nbr))
            except ValueError:
                cg[l_dist][nbr_dist]['contributing_edges'].remove((nbr,n))
            cg[l_dist][nbr_dist]['shared_perim'] -=
                G[n][nbr]["shared_perim"]
            if cg[l_dist][nbr_dist]['contributing_edges'] == []: #if
                districts l_dist and e_dist are no longer adjacent
                 cg.remove_edge(e_dist, l_dist)
        else: raise RuntimeError(f"It should be true that district
            {l_dist} neighbors {e_dist}.")



    elif nbr_dist == l_dist:  #If nbr is part of the l_dist
        #G[n][nbr]["Dist_Boundary"] = 1
        cg.nodes[e_dist]["total_perim"] += G[n][nbr]["shared_perim"]
        cg.nodes[l_dist]["total_perim"] += G[n][nbr]["shared_perim"]
        if (e_dist, nbr_dist) in cg.edges:
            cg[e_dist][nbr_dist]['contributing_edges'].append((n,nbr))
            cg[e_dist][nbr_dist]['shared_perim'] +=
                G[n][nbr]["shared_perim"]
        else: #If this creates a new district boundary
            cg.add_edge(e_dist, nbr_dist)
            cg[e_dist][nbr_dist]['contributing_edges'] = [(n,nbr)]
            cg[e_dist][nbr_dist]['shared_perim'] =
                G[n][nbr]["shared_perim"]

    else:  #Neighboring district is not e_dist or l_dist
        #G[n][nbr]["Dist_Boundary"] = 1
        cg.nodes[e_dist]["total_perim"] += G[n][nbr]["shared_perim"]
        cg.nodes[l_dist]["total_perim"] -= G[n][nbr]["shared_perim"]
        if (l_dist, nbr_dist) in cg.edges:
            try:
                cg[l_dist][nbr_dist]['contributing_edges'].remove((n,nbr))
            except ValueError:
                cg[l_dist][nbr_dist]['contributing_edges'].remove((nbr,n))
            cg[l_dist][nbr_dist]['shared_perim'] -=
                G[n][nbr]["shared_perim"]
            if cg[l_dist][nbr_dist]['contributing_edges'] == []: #if
                districts l_dist and nbr_dist are no longer adjacent
                 cg.remove_edge(l_dist, nbr_dist)
        else: raise RuntimeError(f"It should be true that district
            {l_dist} neighbored {nbr_dist}.")

        if (e_dist, nbr_dist) in cg.edges:
            cg[e_dist][nbr_dist]['contributing_edges'].append((n,nbr))
            cg[e_dist][nbr_dist]['shared_perim'] +=
                G[n][nbr]["shared_perim"]
        else: #If this creates a new district boundary
            cg.add_edge(e_dist, nbr_dist)
            cg[e_dist][nbr_dist]['contributing_edges'] = [(n,nbr)]
            cg[e_dist][nbr_dist]['shared_perim'] =
                G[n][nbr]["shared_perim"]

if cg.nodes[l_dist]["total_perim"] < 0: raise RuntimeError("perimeter
    cannot be negative")
if cg.nodes[e_dist]["total_perim"] < 0: raise RuntimeError("perimeter
    cannot be negative")

cg.nodes[e_dist]['boundary_perim'] += G.nodes[n]['boundary_perim']
cg.nodes[l_dist]['boundary_perim'] -= G.nodes[n]['boundary_perim']

cg.nodes[e_dist]['Reg_Voters'] += G.nodes[n]['Reg_Voters']
cg.nodes[l_dist]['Reg_Voters'] -= G.nodes[n]['Reg_Voters']

cg.nodes[e_dist]["POPULATION"] += G.nodes[n]["POPULATION"]  #Adds
    population to the cg.nodes population entry corresponding to dist
cg.nodes[l_dist]["POPULATION"] -= G.nodes[n]["POPULATION"]  #Subtracts
    population from the cg.nodes population entry corresponding to
    l_dist

cg.nodes[e_dist]["area"] += G.nodes[n]["area"]  #Adds area to the
    cg.nodes area entry corresponding to dist
cg.nodes[l_dist]["area"] -= G.nodes[n]["area"]  #Subtracts area from
    the cg.nodes area entry corresponding to l_dist

cg.nodes[e_dist]["PresRed"] += G.nodes[n]["PresRed"]  #Adds red votes
    to the cg.nodes red votes entry corresponding to dist
cg.nodes[l_dist]["PresRed"] -= G.nodes[n]["PresRed"]  #Subtracts red
    votes from the cg.nodes red votes entry corresponding to l_dist

cg.nodes[e_dist]["PresBlue"] += G.nodes[n]["PresBlue"]  #Adds blue
    votes to the cg.nodes blue votes entry corresponding to dist
cg.nodes[l_dist]["PresBlue"] -= G.nodes[n]["PresBlue"]  #Subtracts blue
    votes from the cg.nodes blue votes entry corresponding to l_dist

cg.nodes[e_dist]["PresOther"] += G.nodes[n]["PresOther"]  #Adds blue
    votes to the cg.nodes blue votes entry corresponding to dist
cg.nodes[l_dist]["PresOther"] -= G.nodes[n]["PresOther"]  #Subtracts
    blue votes from the cg.nodes blue votes entry corresponding to
    l_dist

cg.CDI_matrix[l_dist] = np.zeros((1, ip.num_counties), dtype=np.int16)
    #resets the CDI matrix in row l_dist
cg.CDI_matrix[e_dist] = np.zeros((1, ip.num_counties), dtype=np.int16)
    #resets the CDI matrix in row d_list[1]
```

```
for d in d_list: #'d' represents a district
    cg.nodes[d]['pp_compactness'] = 4 * math.pi * cg.nodes[d]['area'] /
        (cg.nodes[d]['total_perim'] ** 2)
    cg.nodes[d]['shifted_pp_compactness'] = 1 -
        cg.nodes[d]['pp_compactness']
    cg.nodes[d]['total_Pvotes'] = cg.nodes[d]['PresBlue'] +
        cg.nodes[d]['PresRed'] + cg.nodes[d]['PresOther']
    cg.nodes[d]['total_Svotes'] = cg.nodes[d]['SenBlue'] +
        cg.nodes[d]['SenRed'] + cg.nodes[d]['SenOther']
    cg.nodes[d]['blue_Pshare'] = cg.nodes[d]['PresBlue'] /
        cg.nodes[d]['total_Pvotes'] if cg.nodes[d]['total_Pvotes']>0
        else 0
    cg.nodes[d]['red_Pshare'] = cg.nodes[d]['PresRed'] /
        cg.nodes[d]['total_Pvotes'] if cg.nodes[d]['total_Pvotes']>0
        else 0
    cg.nodes[d]['other_Pshare'] = cg.nodes[d]['PresOther'] /
        cg.nodes[d]['total_Pvotes'] if cg.nodes[d]['total_Pvotes']>0
        else 0
    cg.nodes[d]['blue_Sshare'] = cg.nodes[d]['SenBlue'] /
        cg.nodes[d]['total_Svotes'] if cg.nodes[d]['total_Svotes']>0
        else 0
    cg.nodes[d]['red_Sshare'] = cg.nodes[d]['SenRed'] /
        cg.nodes[d]['total_Svotes'] if cg.nodes[d]['total_Svotes']>0
        else 0
    cg.nodes[d]['other_Sshare'] = cg.nodes[d]['SenOther'] /
        cg.nodes[d]['total_Svotes'] if cg.nodes[d]['total_Svotes']>0
        else 0
    cg.nodes[d]['win_threshold_P'] =
        math.ceil(cg.nodes[d]['total_Pvotes']/2 + 0.5)
    cg.nodes[d]['win_threshold_S'] =
        math.ceil(cg.nodes[d]['total_Svotes']/2 + 0.5)
    cg.nodes[d]['Rwasted_votes_P'] = cg.nodes[d]['PresRed'] -
        cg.nodes[d]['win_threshold_P'] if cg.nodes[d]['PresRed'] >
        cg.nodes[d]['PresBlue'] else cg.nodes[d]['PresRed']
    cg.nodes[d]['Bwasted_votes_P'] = cg.nodes[d]['PresBlue'] -
        cg.nodes[d]['win_threshold_P'] if cg.nodes[d]['PresRed'] <=
        cg.nodes[d]['PresBlue'] else cg.nodes[d]['PresBlue']
    for gu in cg.nodes[d]['GUs']:
        county = G.nodes[gu]['county_num']
        cg.CDI_matrix[d][county] += 1

cg.pop_dev_sum = round(sum(abs(p-cg.ideal_pop) for p in
    dict(cg.nodes("POPULATION")).values()))
cg.pop_dev_max = round(max(abs(p-cg.ideal_pop) for p in
    dict(cg.nodes("POPULATION")).values()))
cg.exc_county_splits = np.count_nonzero(cg.CDI_matrix) -
    max(ip.num_counties, ip.num_districts)
cg.exc_gus = sum(np.sum(cg.CDI_matrix, axis=0) - np.max(cg.CDI_matrix,
    axis=0))
cg.cs = cg.exc_county_splits
cg.egu = cg.exc_gus
cg.pp_comp = round(sum([cg.nodes[d]['pp_compactness'] for d in
    cg.nodes]) / cg.number_of_nodes(), 3)
cg.shifted_pp_comp = 1-cg.pp_comp
```

```
    bsP_list = [cg.nodes[d]['blue_Pshare'] for d in cg.nodes]  # bsP = blue
        share for president
    bsP_list = sorted(bsP_list)
    if ip.num_districts % 2 == 0:  # even
        bsP_median = (bsP_list[int(ip.num_districts/2)] +
            bsP_list[ip.num_districts/2-1])/2
    else:  # odd
        bsP_median = bsP_list[int(ip.num_districts/2-0.5)]
    bsP_mean = sum(bsP_list)/ip.num_districts
    cg.mm = abs(bsP_median - bsP_mean)  # median-mean
    bwv_P = sum([cg.nodes[d]['Bwasted_votes_P'] for d in cg.nodes])
    rwv_P = sum([cg.nodes[d]['Rwasted_votes_P'] for d in cg.nodes])
    cg.total_Pvotes = sum([cg.nodes[d]['total_Pvotes'] for d in cg.nodes])
    cg.eg = abs((bwv_P - rwv_P)/cg.total_Pvotes)  # absolute efficiency gap
    cg.all_metrics = [cg.pop_dev_sum, cg.shifted_pp_comp, cg.eg, cg.mm,
        cg.cs, cg.egu]
    cg.metric_keys = ['dpop', 'comp', 'eg', 'mm', 'cs', 'egu']
    cg.metrics = [m for idx, m in enumerate(cg.all_metrics) if
        ip.metrics[cg.metric_keys[idx]] == True]

    #Place these metrics in G as well
    if ip.metrics['dpop']: G.pop_dev[gen][pop_id] = cg.pop_dev_sum
    if ip.metrics['comp']: G.comp[gen][pop_id] = cg.shifted_pp_comp
    if ip.metrics['eg']: G.eg[gen][pop_id] = cg.eg
    if ip.metrics['mm']: G.mm[gen][pop_id] = cg.mm
    if ip.metrics['cs']: G.cs[gen][pop_id] = cg.cs
    if ip.metrics['egu']: G.egu[gen][pop_id] = cg.egu
    G.metrics[gen][pop_id] = cg.metrics
    return cg


def pop_improves(G, dist_graph, n, dist, gen, pop_id):
    """Determines whether population deviation improves if node n is moved
        into district 'dist'.
    G: underlying GU graph
    dg: the district graph. dg = dist_graph[gen][pop_id] = <Graph with d
        nodes>
    n: node that is being moved
    dist: district that n is being moved to
    gen: current generation
    pop_id: population ID"""
    n_dist = G.nodes[n]['district'][gen][pop_id]
    ideal_pop = dist_graph[gen][pop_id].ideal_pop
    if dist_graph[gen][pop_id].nodes[n_dist]["POPULATION"] > ideal_pop and
        dist_graph[gen][pop_id].nodes[dist]["POPULATION"] < ideal_pop:
        return True
    else:
        return False

def comp_improves(G, dist_graph, n, dist, gen, pop_id):
    """Determines whether compactness improves if node n is moved into
        district 'dist'.
    G: underlying GU graph
    dg: the district graph. dg = dist_graph[gen][pop_id] = <Graph with d
        nodes>
    n: node that is being moved
    dist: district that n is being moved to
    gen: current generation
    pop_id: population ID"""
    n_dist = G.nodes[n]['district'][gen][pop_id]
    A1old = dist_graph[gen][pop_id].nodes[n_dist]['area'] #area for
        district 'n_dist' before the flip
    A2old = dist_graph[gen][pop_id].nodes[dist]['area'] #area for district
        'dist' before the flip
    r1old = dist_graph[gen][pop_id].nodes[n_dist]['total_perim'] #perimeter
        for district 'n_dist' before the flip
    r2old = dist_graph[gen][pop_id].nodes[dist]['total_perim'] #perimeter
        for district 'dist' before the flip

    A1new = A1old
    A2new = A2old
    r1new = r1old
    r2new = r2old

    A1new -= G.nodes[n]['area'] #area for district 'n_dist' after the flip
    A2new += G.nodes[n]['area'] #area for district 'dist' after the flip

    for nbr in G.neighbors(n):  #Cycles through neighboring nodes to adjust
        boundary status, perimeter, and district neighbors
        nbr_dist = G.nodes[nbr]['district'][gen][pop_id]
        if nbr_dist == dist:
            #G[n][nbr]["Dist_Boundary"] = 0
            r1new -= G[n][nbr]["shared_perim"]  #This is now 'interior'
                perimeter
            r2new -= G[n][nbr]["shared_perim"]  #Neither n nor nbr is in
                n_dist

        elif nbr_dist == n_dist:  #If nbr is part of the n_dist
            #G[n][nbr]["Dist_Boundary"] = 1
            r1new += G[n][nbr]["shared_perim"]
            r2new += G[n][nbr]["shared_perim"]

        else:  #Neighboring district is not dist or n_dist
            #G[n][nbr]["Dist_Boundary"] = 1
            r2new += G[n][nbr]["shared_perim"]
            r1new -= G[n][nbr]["shared_perim"]
    if A1new/(r1new**2) + A2new/(r2new**2) > A1old/(r1old**2) +
        A2old/(r2old**2): #This signifies that compactness improves
        return True
    else: #This signifies that compactness worsens or stays the same
        return False




def vns(rng, G, ip, gen, pop_id, nbr_graphs, dist_graph):
    """Variable Neighborhood Search. Does a small perturbation on the graph
        for improvement.
    rng: random number generator for code repeatability
    G: base graph
```

```
ip: input
gen: current generation
pop_id: population ID
nbr_graphs: the k-layer neighbor graph dictionary for all nodes.
    nbr_graphs[n] = [list of nodes k away from n]
dist_graph: the array of district graphs. dist_graph[gen][pop_id] =
    <Graph with d nodes>
structure: integer, 1 through 4. Each one does a different type of
    perturbation.
    1: Flip for population improvement
    2: Flip for compactness improvement
    3: Swap for population improvement
    4: Swap for compactness improvement"""
for structure in [1,2]:
    while True: #goes until a successful flip is made
        while True: #Chooses two neighboring districts with populations
            on opposite sides of the ideal population line
            d1 = rng.randint(0,ip.num_districts-1)
            d2 = rng.choice(list(dist_graph[gen][pop_id].neighbors(d1)))
            d1_pop = dist_graph[gen][pop_id].nodes[d1]["POPULATION"]
            d2_pop = dist_graph[gen][pop_id].nodes[d2]["POPULATION"]
            if
                np.sign(d1_pop-ip.ideal_pop)*np.sign(d2_pop-ip.ideal_pop)
                <= 0:
                while d1_pop < d2_pop: #WLOG, make d1 the district with
                    more population
                    temp = d1
                    d1=d2
                    d2=temp
                    temp_pop = d1_pop
                    d1_pop = d2_pop
                    d2_pop = temp_pop
                break #breaks only if dists are on opposite sides of
                    the population balance line

        #Chooses a random edge that makes up the d1-d2 border
        (n,nbr) = rng.choice(dist_graph[gen][pop_id]
            .edges[(d1,d2)]['contributing_edges'])
        if n not in dist_graph[gen][pop_id].nodes[d1]["GUs"]: #Sets n
            as the node in d1
            temp_node = n
            n = nbr
            nbr = temp_node

        # rng.shuffle(dist_graph[gen][pop_id].nodes[d1]['GUs'])
        # n_list_d1 = dist_graph[gen][pop_id].nodes[d1]['GUs']
        # break_flag=False
        # for n in n_list_d1: #Choose two neighboring GUs on the d1-d2
            border
        #     for nbr in G.neighbors(n):
        #         nbr_dist = G.nodes[nbr]['district'][gen][pop_id]
        #         if nbr_dist == d2:
        #             break_flag=True
        #             break
        #     if break_flag == True: break
```

```
            flip_status = flip(G, ip, n, d2, gen, pop_id, nbr_graphs,
                ip.radius, dist_graph, structure)
            if flip_status == True: break #otherwise, we restart the search
                for a new flip
        G, dist_graph = hill_climbing(rng, G, ip, gen, pop_id, nbr_graphs,
            dist_graph, structure)
    return G, dist_graph

            # n = rng.choice(G.nodes)
            # n_dist = G.nodes[n]['district'][gen][pop_id]
            # nbr_dists = set([G.nodes[nbr]['district'][gen][pop_id] for
                nbr in G.neighbors(n)])
            # nbr_dists = nbr_dists - {n_dist}
            # if len(nbr_dists) > 0: #n is on a district border
            #     new_dist = rng.choice(list(nbr_dists))
            # flip(G, ip, n, new_dist, gen, pop_id, nbr_graphs, ip.radius)
                ###Need to fix this. check for whether flip will improve
                population

def checkmetrics(dg, ip):
    """Checks whether the metrics of distgraph are within acceptable
        parameters"""
    if ip.metrics['dpop']:
        if dg.pop_dev_sum > ip.ub["dpop"]: #Acceptable pop_dev allows an
            average district to be 40% off from ideal
            return False, 'dpop', dg.pop_dev_sum
    if ip.metrics['comp']:
        #if dg.shifted_pp_comp > 0.8888: #Acceptable average compactness is
            less than 0.8888
        if dg.shifted_pp_comp > ip.ub["comp"]:
            return False, 'comp', dg.shifted_pp_comp
    if ip.metrics['eg']:
        if dg.eg > ip.ub["eg"]: #Acceptable efficiency gaps are 3*8% (>8%
            is considered problematic by Stephanopoulos and McGhee)
            return False, 'eg', dg.eg
    if ip.metrics['mm']:
        if dg.mm > ip.ub["mm"]: #Acceptable median-mean calculations allow
            <= 0.05
            return False, 'mm', dg.mm
    if ip.metrics['cs']:
        if dg.cs > ip.ub["cs"]: #Acceptable county-split count is 50 or
            fewer
            return False, 'cs', dg. cs
    if ip.metrics['egu']:
        if dg.egu > ip.ub["egu"]: #Acceptable excess gu count is 500 or
            fewer
            return False, 'egu', dg.egu
    #If we've made it this far, then all metrics are okay.
    return True, ":)", math.pi

def hill_climbing(rng, G, ip, gen, pop_id, nbr_graphs, dist_graph,
    structure):
    """Searches all nodes to see if any flips will improve the map.
    rng: random number generator for code repeatability
    G: base graph
```

```
ip: input
gen: current generation
pop_id: population ID of current plan
nbr_graphs: the k-layer neighbor graph dictionary for all nodes.
    nbr_graphs[n] = [list of nodes k away from n]
dist_graph: the array of district graphs. dist_graph[gen][pop_id] =
    <Graph with d nodes>"""
#1. Make a flip
#2. Create new dist_graph for the flip
#3. See if new plan dominates old plan. If so, accept the flip.
#4. Repeat until all nodes are checked, or until a certain fraction of
    nodes are checked
hill_climbing_count = 1 #temp
hc_cycles = 0 #hill climbing cycles. The number of times hill climbing
    was performed
G2 = copy.deepcopy(G) #copy of G to make edits to
##curr_dg = copy.deepcopy(dist_graph[gen][pop_id])
curr_dist_graph = copy.deepcopy(dist_graph)
while hill_climbing_count >0: #Only continues as long as we're making
    progress
    if hc_cycles >= 10: break #breaks if hill climbing is taking too
        long
    hc_cycles += 1
    hill_climbing_count = 0
    for n in G2.nodes:
        n_dist = G2.nodes[n]['district'][gen][pop_id]
        nbr = check_if_on_dist_boundary(rng, G2, gen, pop_id, n)
        if nbr != None: #if n is on a district boundary
            nbr_dist = G2.nodes[nbr]['district'][gen][pop_id]
            curr_metrics = curr_dist_graph[gen][pop_id].metrics
            flip_status = flip(G2, ip, n, nbr_dist, gen, pop_id,
                nbr_graphs, ip.radius, curr_dist_graph, structure)
            if flip_status == True: #if the flip succeeded
                candidate_metrics = curr_dist_graph[gen][pop_id].metrics
                if dominates_metrics(candidate_metrics, curr_metrics,
                    structure):
                    hill_climbing_count += 1
                else:
                    G2.nodes[n]['district'][gen][pop_id] = n_dist
                        #flips back if not dominant
                    upd_dg(G2, ip, curr_dist_graph, gen, pop_id, n,
                        nbr_dist, n_dist)
                    #fs2 = flip(G2, ip, n, n_dist, gen, pop_id,
                        nbr_graphs, float('inf'), curr_dist_graph)
                        #flips back if not dominant
                    #if fs2 == False: raise RuntimeError("fs2 shouldn't
                        be False. We are flipping *back*")
if dominates(curr_dist_graph[gen][pop_id], dist_graph[gen][pop_id]):
    dist_graph[gen][pop_id] = curr_dist_graph[gen][pop_id]
    G = G2
    print(f"Gen {gen}. pop_id={pop_id}. Finished hill climbing using
        structure {structure}. Completed {hc_cycles} hill climbing
        cycles. ")
else:
        print(f"Gen {gen}. pop_id={pop_id}. Finished hill climbing using
            structure {structure}. No improvement after completing
            {hc_cycles} hill climbing cycles.")
    return G, dist_graph



def check_if_on_dist_boundary(rng, G, gen, pop_id, n, d2=None):
    """Checks if a node is on a district boundary. Returns nbr if it is.
       Returns None if it not on a district boundary.
    G: base graph
    gen: current generation
    pop_id: population ID of current plan
    n: node to check
    d2: (optional) the specific district that we want to see if n borders"""
    n_dist = G.nodes[n]['district'][gen][pop_id]
    if d2 == n_dist: raise ValueError(f"d2 ({d2}) cannot be equal to n_dist
       ({n_dist}).")
    nbrs = list(G.neighbors(n))
    rng.shuffle(nbrs)
    for nbr in nbrs:
        nbr_dist = G.nodes[nbr]['district'][gen][pop_id]
        if d2 == None and nbr_dist != n_dist:
            return nbr
        elif d2 != None and nbr_dist == d2:
            return nbr
        #else we continue until a proper nbr is found
    #If we made it this far, then no nbr is is a different district
    return None


def nondominated_sorting(dist_graph_list):
    """Sorts the entries of dist_graph_list into fronts
    dist_graph_list = list of district graphs"""
    dgl = dist_graph_list #An alias
    dominated_solns = [] #This will contain the dominated plans.
    nondom = [] #This will contain the nondominated plans.
    nondom_idxs = [] #This will contain the indices of the nondominated
       plans.
    rank=0 #This will denote the front the plan is assigned to
    fronts = [] #List of lists. Each entry will be the list of plans in
       that front
    fronts_idxs = [] #List of lists. Each entry will the list of plan
       indices in that front
    plan_ids = {}
    for i,dg in enumerate(dgl): plan_ids[dg] = i #This dictionary assigns
       each dist_graph dg to an index i
    #Loop for first front
    for dg1 in dgl:
        for dg2 in dgl:
            if dominates(dg2, dg1): #dg2 dominates dg1
                dominated_solns.append(dg1)
                break #no more calculations in second 'for' loop are needed
                   since we've determined that dg1 is dominated
    nondom = list(set(dgl) - set(dominated_solns))
    fronts.append(nondom)
    for dg in nondom: nondom_idxs.append(plan_ids[dg])
    fronts_idxs.append(nondom_idxs)

    #Loop for every subsequent front
    while dominated_solns != []:
        rank += 1 #helpful for debugging
        temp_domsols = []
        nondom_idxs = [] #This needs to reset every loop
        for dg1 in dominated_solns:
            for dg2 in dominated_solns:
                if dominates(dg2, dg1): #dg2 dominates dg1
                    temp_domsols.append(dg1)
                    break
        nondom = list(set(dominated_solns) - set(temp_domsols))
        fronts.append(nondom)
        for dg in nondom: nondom_idxs.append(plan_ids[dg])
        fronts_idxs.append(nondom_idxs)
        dominated_solns = temp_domsols #makes a (shallow) copy
        del temp_domsols #Deletes the temporary variable
    return fronts_idxs, fronts

def dominates(solution1, solution2):
    """Check if solution1 dominates solution2. 'solution1' and 'solution2'
       are graphs"""
    s1_mlist = solution1.metrics
    s2_mlist = solution2.metrics
    doms = all(s1 <= s2 for s1, s2 in zip(s1_mlist, s2_mlist)) and any(s1 <
       s2 for s1, s2 in zip(s1_mlist, s2_mlist))
    return doms

def dominates_metrics(solution1, solution2, structure=None):
    """Check if solution1 dominates solution2. 'solution1' and 'solution2'
       are lists of metric values"""
    if structure == None:
        doms = all(s1 <= s2 for s1, s2 in zip(solution1, solution2)) and
           any(s1 < s2 for s1, s2 in zip(solution1, solution2))
    elif structure == 1:
        doms = solution1[0] < solution2[0]
    elif structure == 2:
        doms = solution1[1] < solution2[1]
    return doms

def assign_next_gen(G, ip, gen, front, dist_graph, min_pop_id):
    """Assigns the best <ip.pop_size> plans to the next generation
    G: Graph
    ip: input
    gen: the *next* generation to be assigned
    front: the list of plans in a particular rank
    dist_graph: the array of district graphs. dist_graph[gen][pop_id] =
       <networkx graph>
    min_pop_id: the smallest pop_id that will be assigned in this iteration
       of the function"""
    for idx, dg in enumerate(front): #dg = dist_graph
```

```
        pop_id = min_pop_id + idx
        if ip.metrics['dpop']: G.pop_dev[gen][pop_id] = dg.pop_dev_sum
        if ip.metrics['comp']: G.comp[gen][pop_id] = dg.shifted_pp_comp
        if ip.metrics['eg']: G.eg[gen][pop_id] = dg.eg
        if ip.metrics['mm']: G.mm[gen][pop_id] = dg.mm
        if ip.metrics['cs']: G.cs[gen][pop_id] = dg.cs
        if ip.metrics['egu']: G.egu[gen][pop_id] = dg.egu
        G.metrics[gen][pop_id] = dg.metrics
        dist_graph[gen][pop_id] = dg

        node_count = 0
        for dist in dg.nodes:
            for gu in dg.nodes[dist]['GUs']:
                G.nodes[gu]['district'][gen][pop_id] = dist
                node_count+=1
        if node_count != ip.num_GUs:
            raise ValueError(f"We should have {ip.num_GUs} GUs")

def crowding_distance(front, num_to_keep):
    """Finds the crowding distance for each solution in a given front and
    returns the least crowded solutions up to the num_to_keep
    front: the list of plans that we will calculate the crowding distance
        for"""
    cd = {}  # Crowding distance
    for idx, _ in enumerate(front):
        cd[idx] = 0
    num_metrics = len(front[0].metrics) #determines the number of metrics
        we used
    for i in range(num_metrics):  # 'i' represents the metric index
        soln_list = sorted([(idx, soln.metrics[i]) for idx, soln in
            enumerate(front)], key=lambda x: x[1])  # sorts metric list by
            metric value
        # maxval - minval finds range for the metric
        range_metric = soln_list[-1][1] - soln_list[0][1]
        # 'j' will be used to identify spot in list
        for j, soln in enumerate(soln_list):
            idx = soln[0]  # 'idx' represents the index of the solution
            if j == 0 or j == len(soln_list)-1:
                cd[idx] += float('inf')
            else:
                try:
                    cd[idx] += (soln_list[j+1][1] - soln_list[j-1][1]) /
                        range_metric
                except ZeroDivisionError:
                    cd[idx] += (soln_list[j+1][1] - soln_list[j-1][1]) / 1
    sorted_idxs = sorted([(idx, val) for idx, val in cd.items()],
        key=lambda x: x[1], reverse=True)
    sorted_front = [front[i[0]] for i in sorted_idxs]
    #sorted_pop_ids = [front_pop_id[i[0]] for i in sorted_idxs]
    #return sorted_pop_ids, sorted_front[0:num_to_keep]  # Keeps the best
        'num_to_keep' graphs
    return sorted_front[0:num_to_keep]  # Keeps the best 'num_to_keep'
        graphs
```

```
def plot_plans(G, ip, county_lines, gen, min_idx=0, max_idx=None):
    """Plots all maps between min_idx and max_idx
    G: base graph
    ip: input
    county_lines: Graph that will be used to plot the county lines
    gen: the generation to plot. Defaults to last generation (i.e., gen=-1)
    min_idx: the smallest pop_id to plot. Defaults to first plan (i.e.,
        min_idx=0)
    max_idx: the largest pop_id to plot. Defaults to None. Code will
        automatically plot all plans in generation gen in this case. """
    high_contrast_colors = ['blue', 'green', 'yellow', 'magenta', 'cyan',
        'orange', 'purple']
    cmap = mcolors.ListedColormap(high_contrast_colors)
    for pop_id in range(2*ip.pop_size):
        if G.nodes[0]['district'][gen][pop_id] != -1:
            G.data[f'plan{pop_id}'] = [G.nodes[n]['district'][gen][pop_id]
                for n in G.nodes()]

    if max_idx == None:
        i=min_idx
        planx = 'plan' + f'{i}'
        while planx in G.data:
            fig, ax = plt.subplots()
            G.data.plot(column=planx, ax=ax, legend=True, cmap=cmap)
            county_lines.data.boundary.plot(ax=ax, color="black")
            ax.set_title(f"Plan {i}")
            fig.show()
            i += 1
            planx = 'plan' + f'{i}'
    else:
        for i in range(min_idx, max_idx):
            planx = 'plan' + f'{i}'
            fig, ax = plt.subplots()
            G.data.plot(column=planx, ax=ax, legend=True, cmap=cmap)
            county_lines.data.boundary.plot(ax=ax,color="black")
            ax.set_title(f"Plan {i}")
            fig.show()

def hypervolume(ip, gen_col2, front_col, metlist):
    """Calculates the hypervolume columns of values given a graph G
    metlist[0] will contain all scaled pop_dev values; metlist[1] might
        contain all scaled
    compactness values
    ip: input
    gen_col2: if all plans are placed in a line, this would be the
        generation value for each plan
    front_col: contains the ranks for each plan
    metlist: contains the list of metric values for each plan"""
    front0 = [[] for _ in range(ip.num_generations+1)]
    #front0 = np.empty(shape=(0, ip.num_generations))
    volume = []
    ref = [1.1 for _ in range(len(metlist))]
    for row in range(len(metlist[0])):
        gen = gen_col2[row]
        if front_col[row] == 0: #If the entry is in rank 0
```

```
            front0[gen].append([metlist[i][row] for i in
                range(len(metlist))])
            #front0[gen] = [metlist[i][row] for i in range(len(metlist))]
    for gen in range(len(front0)):
        front0[gen] = np.array(front0[gen])
    #front0arr = np.array(front0)
    minval = [[] for _ in range(len(metlist))] #minval[gen][metric] = number
    for gen in range(ip.num_generations+1):
        #hypvol = hv.HyperVolume(ref)
        #volume.append(hypvol.compute(front0[gen]))
        ind = HV(ref_point=ref)
        volume.append(ind(front0[gen]))
    return volume

def plot_hvolume(hvolume):
    """Plots the hypervolume as a line chart.
    hvolume: list of hypervolume values by generation."""
    # Create a figure and a set of subplots
    fig, ax = plt.subplots(nrows=1, ncols=1)

    # Plot on the first subplot
    ax.plot(hvolume)

    # Add a title to the subplot
    ax.set_title("Hypervolume by Generation")

    # Add labels to the axes
    ax.set_xlabel("Generation")
    ax.set_ylabel("Hypervolume")

    # Set x-axis ticks to be integers
    ax.set_xticks(range(len(hvolume)))

    # Display the chart
    fig.show()



def main(*args):
    # ------------------------------------------------------------
    #Main algorithm
    timetxt = datetime.datetime.now().strftime("_%m%d%y_%H%M%S_%f")

    rng = random.Random(args[1]) #seeds 'random'


    if isinstance(args[0], str) == True:
        file_name = args[0]
        ip = input_c.Load_from_file(file_name)
    elif args[0] == None:
        print("No input file provided. Running with default parameters")
        ip = input_c(
            json_file="./SC_Precincts_2_FeaturesToJSO.geojson",
            ps=50,  # population size
            mp=0.05,  # mutation probability
            nd=7,  # number of districts
            ng=3,  # number of generations
            met={  # metrics
                "dpop": True,  # population
                "comp": True,  # compactness
                "eg": True,  # efficiency gap
                "mm": False,  # median-mean
                "cs": True,  # county splits
                "egu": True # excess geographic units
            },
            cf="./SC_Counties_20_FeaturesToJSO.geojson",
            lmn={  # long metric names
                "dpop": "Pop Dev",  # population
                "comp": "Compactness",  # compactness
                "eg": "Eff Gap",  # efficiency gap
                "mm": "Med Mean",  # median-mean
                "cs": "County Splits",  # county splits
                "egu": "Excess GUs" # excess geographic units
            },
            rnd=1,
            rad=3
        )
    ip.rand_seed = args[1]
    ip.ub = {} # Upper bounds for the various metrics
    ip.ub["dpop"] = None # Initialize this in the next step
    ip.ub["comp"] = 0.9 #Corresponds to invPP of ?? (not 10 anymore)
    ip.ub["eg"] = 0.24
    ip.ub["mm"] = 0.05
    ip.ub["cs"] = 50
    ip.ub["egu"] = 500
    print(f"Objectives are {[met for met in ip.metrics.keys() if
        ip.metrics[met] == True]}")
    gen = 0
    start = time.time()
    # ---------------------------------------------------------------
    # Initialize Graph
    stateG = initialize_graph(ip)
    dist_graph = [[None for _ in range(ip.pop_size*2)] for _ in
        range(ip.num_generations+1)]
    ip.ub["dpop"] = ip.num_districts*target_pop(stateG, ip)*0.4

    nbr_graphs = make_nbr_graphs(stateG, ip.radius)

    print("Initialized graph")
    cp0 = time.time()

    # ---------------------------------------------------------------
    # Create attribute graph (for counties)
    countygraph = create_attr_graph(stateG, ip, attr="county_num")
    print("Finished county graph")
    ip.num_counties = countygraph.number_of_nodes()
    cp1 = time.time()

    # ---------------------------------------------------------------
    # Create shortest paths dictionary
    try:
        with open('sp.pkl', 'rb') as fp: shortest_paths = pickle.load(fp)
    except FileNotFoundError:  # If we haven't done this before
        shortest_paths = nx.shortest_path(stateG)
        with open('sp.pkl', 'wb') as fp: pickle.dump(shortest_paths, fp)
    print("Created sp dictionary")
    cp2 = time.time()

    # ----------------------------------------------------------------
    # Create shortest path lengths dictionary
    try:
        with open('spl.pkl', 'rb') as fp: spl = pickle.load(fp)
    except FileNotFoundError:  # If we haven't done this before
        spl = np.zeros([len(shortest_paths), len(shortest_paths)],
            dtype=int)  # Shortest path lengths
        for i in range(len(shortest_paths)):
            for j in range(len(shortest_paths)):
                spl[i][j] = len(shortest_paths[i][j])
        with open('spl.pkl', 'wb') as fp: pickle.dump(spl, fp)
    print("Created spl dictionary")
    cp3 = time.time()



    # ----------------------------------------------------------------
    # Initialize front_col2 (This contains the rank for each map)
    front_col2 = [None]*(ip.num_generations+1)*ip.pop_size*2

    # ----------------------------------------------------------------
    # Initialize first <ip.pop_size> plans
    for pop_id in range(ip.pop_size):
        generate_plan(rng, stateG, ip, 0, pop_id, randomness=ip.randomness)
        dist_graph[gen][pop_id] = create_attr_graph(stateG, ip,
            gen_id=(gen, pop_id))
    cp4 = time.time()

    # ----------------------------------------------------------------
    # Initialize Time variables
    time_initialize_graphs = cp0-start
    time_create_attr_graphs = cp1-cp0
    time_shortest_paths = cp2-cp1
    time_spl = cp3 - cp2
    time_initialize_plans = cp4-cp3
    time_crossover = 0
    time_mutation = 0
    time_copying = 0
    time_dist_graphs = 0
    time_vns = 0
    time_nondom_sort = 0
    time_crowd_dist = 0

    # ----------------------------------------------------------------
    # Determine how much of population is found with crossover, mutation,
        and copying
```

```
num_crossover = round(0.6*ip.pop_size)
num_mutation = round(0.3*ip.pop_size)
num_copy = ip.pop_size - num_crossover - num_mutation

# ----------------------------------------------------------------
# Main loop
while gen < ip.num_generations:

    # ----------------------------------------------------------------
    # #Crossover
    # ##for pop_id_idx in range(round(ip.pop_size/2+0.01)): #Halved bc
       2 children created, +0.01 to get correct rounding
    # for pop_id_idx in range(ip.pop_size):
    #     #Choose crossover point
    #     c = rng.randint(0,stateG.number_of_nodes()-1)

    #     #Choose 2 random parents
    #     p1 = rng.randint(0, ip.pop_size-1)
    #     p2 = p1 #temp
    #     while p2 == p1: p2 = rng.randint(0, ip.pop_size-1)

    #     crossover(random, stateG, ip, gen, nbr_graphs, pop_id_idx,
       p1, p2, c, single_child=True) #will create a child
    # cp5 = time.time()
    # time_crossover += cp5-cp4_

    # # ----------------------------------------------------------------
    # # Mutation
    # for pop_id in range(ip.pop_size, 2*ip.pop_size):
    #     mutation(random, stateG, ip, gen, pop_id, nbr_graphs)
    # cp6 = time.time()
    # time_mutation += cp6-cp5

    for pop_id in range(ip.pop_size, 2*ip.pop_size):

        objs_satisfactory = False #Flag for whether all objectives fall
            under UBs
        while objs_satisfactory == False:
            #Crossover
            if pop_id in range(ip.pop_size, ip.pop_size+num_crossover):
                cp4_ = time.time()
                c = rng.randint(0,stateG.number_of_nodes()-1)

                #Choose 2 parents through tournament selection
                p1 = tournament_selection(random, ip, dist_graph, gen,
                    3)
                p2 = p1 #temp
                while p2 == p1: p2 = tournament_selection(random, ip,
                    dist_graph, gen, 3)
                crossover(random, stateG, ip, gen, nbr_graphs, pop_id,
                    p1, p2, c, single_child=True)
                #print(f"Gen {gen}. Crossover for plan {pop_id}.")
                cp5_0 = time.time()
                last_cp = cp5_0
                time_crossover += cp5_0-cp4_
            #Mutation
            elif pop_id in range(ip.pop_size+num_crossover,
                ip.pop_size+num_crossover+num_mutation):
                cp4_ = time.time()
                p = tournament_selection(random, ip, dist_graph, gen,
                    3) #parent to copy from
                dist_graph[gen][pop_id] =
                    copy.deepcopy(dist_graph[gen][p])
                for n in stateG.nodes:
                    stateG.nodes[n]['district'][gen][pop_id] =
                        stateG.nodes[n]['district'][gen][p]
                mutation(random, stateG, ip, gen, pop_id, nbr_graphs)
                #print(f"Gen {gen}. Mutation for plan {pop_id}.")
                cp5_1 = time.time()
                last_cp = cp5_1
                time_mutation += cp5_1-cp4_

            #Copying
            else:
                cp4_ = time.time()
                p = tournament_selection(random, ip, dist_graph, gen,
                    3) #parent to copy from
                dist_graph[gen][pop_id] =
                    copy.deepcopy(dist_graph[gen][p])
                for n in stateG.nodes:
                    stateG.nodes[n]['district'][gen][pop_id] =
                        stateG.nodes[n]['district'][gen][p]
                #print(f"Gen {gen}. Copied for plan {pop_id}.")
                cp5_2 = time.time()
                last_cp = cp5_2
                time_copying += cp5_2-cp4_

            #Make District Graph
            dist_graph[gen][pop_id] = create_attr_graph(stateG, ip,
                gen_id=(gen, pop_id))
            objs_satisfactory, failedmet, metval =
                checkmetrics(dist_graph[gen][pop_id], ip)
            if objs_satisfactory == False: print(f"Gen {gen}. Failed
                districting {pop_id} (metric too large:
                {failedmet}={metval}).")
            else: print(f"Gen {gen}. Successfully created districting
                {pop_id}.")
            cp6 = time.time()
            time_dist_graphs += cp6 - last_cp



    # --------------------------------------------------------------
    # Create District Graphs
    #if gen==0: dg_to_make = range(2*ip.pop_size)
    #else: dg_to_make = range(ip.pop_size, 2*ip.pop_size)
    # dg_to_make = range(ip.pop_size, 2*ip.pop_size)
    # for pop_id in dg_to_make:
```

```
#     dist_graph[gen][pop_id] = create_attr_graph(stateG, ip,
    gen_id=(gen, pop_id))
# cp6_1 = time.time()
# time_dist_graphs += cp6_1 - cp6_0

# ---------------------------------------------------------------
# Variable Neighborhood Search (and Hill Climbing)
num_plans_to_vns = min(round(0.2*ip.pop_size), 5)
pop_ids_to_vns = rng.sample(range(ip.pop_size, 2*ip.pop_size),
    num_plans_to_vns)
for pop_id in pop_ids_to_vns:
    stateG, dist_graph = vns(random, stateG, ip, gen, pop_id,
        nbr_graphs, dist_graph)
cp6_2 = time.time()
time_vns += cp6_2 - cp6

# ---------------------------------------------------------------
# Nondominated Sorting
fronts_pop_id, fronts = nondominated_sorting(dist_graph[gen])
cp7 = time.time()
time_nondom_sort += cp7-cp6_2


for rank, f in enumerate(fronts_pop_id):
    for pop_id in f:
        front_col2[gen*ip.pop_size*2 + pop_id] = rank #Assigns the
            rank (i.e. the front number) that each plan falls in
        dist_graph[gen][pop_id].rank = rank

# ---------------------------------------------------------------
# Crowding Distance and Assigning Next Generation
front_lengths = [len(f) for f in fronts_pop_id]
stateG.front0_sizes.append(front_lengths[0])
cum_lengths = [sum(front_lengths[:i+1]) for i in
    range(len(front_lengths))]
if gen < ip.num_generations-1:
    for idx, l in enumerate(cum_lengths):
        if idx == 0: min_pop_id=0
        else: min_pop_id=cum_lengths[idx-1]

        if l <= ip.pop_size: #If we can fit this entire front in
            the next generation
            assign_next_gen(stateG, ip, gen+1, fronts[idx],
                dist_graph, min_pop_id)
        elif l > ip.pop_size: #If this front is too big to all fit
            in the next generation
            num_to_keep = front_lengths[idx] - cum_lengths[idx] +
                ip.pop_size
            partial_front = crowding_distance(fronts[idx],
                num_to_keep)
            assign_next_gen(stateG, ip, gen+1, partial_front,
                dist_graph, min_pop_id)
            break #Don't assign any more to the next gen
else: #If we are on the last generation, only assign the Pareto
    front
```

```
            assign_next_gen(stateG, ip, gen+1, fronts[0], dist_graph,
                min_pop_id=0)
        cp8 = time.time()
        time_crowd_dist += cp8-cp7


        #Advance the generation
        gen += 1
        curr_time = time.time()
        if curr_time-start > 255600:
            break #breaks if we hit 71-hour time limit

    ip.front0_sizes = stateG.front0_sizes
    end = time.time()
    ip.runtime = end-start

    # ----------------------------------------------------------------
    # Times
    print(f"time for initialize_graph = {time_initialize_graphs:.2f}")
    print(f"time for create_attr_graph = {time_create_attr_graphs:.2f}")
    print(f"time for shortest_paths = {time_shortest_paths:.2f}")
    print(f"time for shortest path lengths = {time_spl:.2f}")
    print(f"time for initialize_plans = {time_initialize_plans:.2f}")
    print(f"time for crossover = {time_crossover:.2f}")
    print(f"time for mutation = {time_mutation:.2f}")
    print(f"time for dist_graphs = {time_dist_graphs:.2f}")
    print(f"time for vns = {time_vns:.2f}")
    print(f"time for nondominated_sorting = {time_nondom_sort:.2f}")
    print(f"time for crowding_distance = {time_crowd_dist:.2f}")
    print(f"total time = {end-start:.2f}")

    #------------------------------------------------------------
    # df0: Starting data
    input_data = [list(ip.__dict__.values())]
    input_columns = list(ip.__dict__.keys())
    df0 = pd.DataFrame(input_data, columns=input_columns)


    #------------------------------------------------------------
    # df1: Assigning Precincts
    cols1 = ['Generation', 'pop_id', 'node', 'district']

    non = stateG.number_of_nodes()
    ps = ip.pop_size
    ng = ip.num_generations
    f0s = stateG.front0_sizes[-1]
    if ng <= 4:
        range_rows = range(non*ps*2*ng + non*f0s)
        range_gen = range(ng+1)
        range_ps = range(ps*2)
        num_to_del = -(2*ps-f0s)*non
    else: #If the number of generations is too large, only record last front
        range_rows = range(non*ps*2*ng,
           non*ps*2*ng+min(front_lengths[0],ps)*non)
        range_gen = range(ng, ng+1)
        range_ps = range(min(front_lengths[0],ps))
        num_to_del = None
    gen_col = [math.floor(i/(non*ps*2)) for i in range_rows]
    id_col = [math.floor(i/non) % (2*ps) for i in range_rows]
    node_col = [(i%non)+1 for i in range_rows]

    district_col = [stateG.nodes[n]['district'][gen][pop_id] for gen in
        range_gen for pop_id in range_ps for n in stateG.nodes]
    district_col = district_col[:num_to_del]

    # -----------------------------------------------------------------------
    # df2: For recording the metrics of each plan
    cols2 = ['Generation', 'pop_id', 'rank'] + [ip.long_met_names[met] for
        met in ip.metrics if ip.metrics[met] == True ]
    num_rows2 = (ng)*ps*2 + stateG.front0_sizes[-1]
    gen_col2 = [math.floor(i/(2*ps)) for i in range(num_rows2)]
    id_col2 = [i % (2*ps) for i in range(num_rows2)]
    for idx in range(stateG.front0_sizes[-1]):
        front_col2[gen*ps*2 + idx] = 0 #Assigns the last generation rank 0
            (because they necessarily must be)


    if ip.metrics['dpop']: pop_col = [stateG.pop_dev[gen][pop_id] for gen
        in range(ng+1) for pop_id in range(ps*2)]
    if ip.metrics['comp']: comp_col = [stateG.comp[gen][pop_id] for gen in
        range(ng+1) for pop_id in range(ps*2)]
    if ip.metrics['eg']: eg_col = [stateG.eg[gen][pop_id] for gen in
        range(ng+1) for pop_id in range(ps*2)]
    if ip.metrics['mm']: mm_col = [stateG.mm[gen][pop_id] for gen in
        range(ng+1) for pop_id in range(ps*2)]
    if ip.metrics['cs']: cs_col = [stateG.cs[gen][pop_id] for gen in
        range(ng+1) for pop_id in range(ps*2)]
    if ip.metrics['egu']: egu_col = [stateG.egu[gen][pop_id] for gen in
        range(ng+1) for pop_id in range(ps*2)]

    #num_to_del2 = 2*ps - max(stateG.front0_sizes[-1], ps)
    num_to_del2 = 2*ps - stateG.front0_sizes[-1]
    if ip.metrics['dpop']: pop_col = pop_col[:-num_to_del2]
    if ip.metrics['comp']: comp_col = comp_col[:-num_to_del2]
    if ip.metrics['eg']: eg_col = eg_col[:-num_to_del2]
    if ip.metrics['mm']: mm_col = mm_col[:-num_to_del2]
    if ip.metrics['cs']: cs_col = cs_col[:-num_to_del2]
    if ip.metrics['egu']: egu_col = egu_col[:-num_to_del2]
    front_col2 = front_col2[:-num_to_del2]

    metlist = []
    if ip.metrics['dpop']: metlist.append([x/ip.ub["dpop"] for x in
        pop_col])
    if ip.metrics['comp']: metlist.append([x/ip.ub["comp"] for x in
        comp_col])
    if ip.metrics['eg']: metlist.append([x/ip.ub["eg"] for x in eg_col])
    if ip.metrics['mm']: metlist.append([x/ip.ub["mm"] for x in mm_col])
    if ip.metrics['cs']: metlist.append([x/ip.ub["cs"] for x in cs_col])
    if ip.metrics['egu']: metlist.append([x/ip.ub["egu"] for x in egu_col])
```

```
hvolume = hypervolume(ip, gen_col2, front_col2, metlist)

# num_to_del3 = -(ps - min(stateG.front0_sizes[-1], ps))
# if num_to_del3 == 0: num_to_del3 = None
# gen_col2 = gen_col2[:num_to_del3]
# id_col2 = id_col2[:num_to_del3]

df1_dict = {} #Contains GU assignments
df1_dict['Generation'] = gen_col
df1_dict['pop_id'] = id_col
df1_dict['node'] = node_col
df1_dict['district'] = district_col

df2_dict = {} #Contains metrics
df2_dict['Generation'] = gen_col2
df2_dict['pop_id'] = id_col2
df2_dict['rank'] = front_col2
if ip.metrics['dpop']: df2_dict['Pop Dev'] = pop_col
if ip.metrics['comp']: df2_dict['Compactness'] = comp_col
if ip.metrics['eg']: df2_dict['Eff Gap'] = eg_col
if ip.metrics['mm']: df2_dict['Med Mean'] = mm_col
if ip.metrics['cs']: df2_dict['County Splits'] = cs_col
if ip.metrics['egu']: df2_dict['Excess GUs'] = egu_col

# ------------------------------------------------------------------------
# df3: Generational data
hvstring = [met for met in list(ip.metrics.keys()) if ip.metrics[met]]
hvstring.insert(0, 'hypervolume')
hvstring = ' '.join(hvstring)
cols3 = [hvstring]
df3_dict = {} #Contains generational measures
df3_dict[hvstring] = hvolume

df1 = pd.DataFrame(data=df1_dict, columns=cols1, dtype=np.int16)
df2 = pd.DataFrame(data=df2_dict, columns=cols2)
df3 = pd.DataFrame(data=df3_dict, columns=cols3)

met_string = ''
for met in ip.metrics:
    if ip.metrics[met]==True: met_string += 'O'
    else: met_string += 'X'

excel_doc_name = "Van_" + met_string + timetxt + ".xlsx"
excel_writer = pd.ExcelWriter(excel_doc_name, engine='xlsxwriter')

df0.to_excel(excel_writer, sheet_name='Input Data')
df1.to_excel(excel_writer, sheet_name='GU Assignment')
df2.to_excel(excel_writer, sheet_name='Metrics')
df3.to_excel(excel_writer, sheet_name='Generational Data')

excel_writer.save()


# --------------------------------------------------------------------
# Graphing
```

```
    try:
        with open('county_boundaries.pkl', 'rb') as fp: county_boundaries =
            pickle.load(fp)
    except FileNotFoundError:
        county_boundaries = Graph.from_file(
            ip.county_file,
            adjacency="rook",
            reproject="True",
            ignore_errors="True"
        )
        with open('county_boundaries.pkl', 'wb') as fp:
            pickle.dump(county_boundaries, fp)

    plot_plans(stateG, ip, county_boundaries, gen)
    plot_hvolume(hvolume)

    print("finished")

if __name__ == "__main__":
    rseed = secrets.randbelow(sys.maxsize)
    #rseed = 1193283797044648948
    random.seed(rseed)
    print("random seed is", rseed)
    if len(sys.argv) == 1:
        main('NSGA2_input3.txt', rseed) #Use file name if reading from file
    elif len(sys.argv) == 2:
        input_file = sys.argv[1]
        main(input_file, rseed)
```

Bibliography

[1] David Abramson, Mohan Krishnamoorthy, Henry Dang, et al. Simulated annealing cooling schedules for the school timetabling problem. *Asia Pacific Journal of Operational Research*, 16:1–22, 1999.

[2] Baker v. Carr. 369 U.S. 186 (1962).

[3] Pietro Belotti, Austin Buchanan, and Soraya Ezazipour. Political districting to optimize the polsby-popper compactness score. *Draft manuscript*, 2023.

[4] John R Birge. Redistricting to maximize the preservation of political boundaries. *Social Science Research*, 12(3):205–214, 1983.

[5] Michelle H Browdy. Simulated annealing: an improved computer model for political redistricting. *Yale Law & Policy Review*, 8(1):163–179, 1990.

[6] Steven J. D'Amico, Shoou-Jiun Wang, Rajan Batta, and Christopher M. Rump. A simulated annealing approach to police district design. *Computers & Operations Research*, 29(6):667–684, 2002.

[7] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002.

[8] Kalyanmoy Deb, Karthik Sindhya, and Jussi Hakanen. Multi-objective optimization. In *Decision sciences*, pages 161–200. CRC Press, 2016.

[9] Daryl DeFord, Moon Duchin, and Justin Solomon. Recombination: A family of markov chains for redistricting. *arXiv preprint arXiv:1911.05725*, 2019.

[10] Matthew Dube and Jesse Clark. Beyond the circle: Measuring district compactness using graph theory. In *Northeast Political Science Association Conference*, 2016.

[11] Moon Duchin. Outlier analysis for pennsylvania congressional redistricting. *LWV vs. Commonwealth of Pennsylvania Docket No. 159 MM 2017*, 2018.

[12] Bernard Grofman and Jonathan Cervas. Recent approaches to the definition and measurement of compactness. *Available at SSRN 3919249*, 2021.

[13] Wes Gurnee and David B Shmoys. Fairmandering: A column generation heuristic for fairness-optimized political districting. In *SIAM Conference on Applied and Computational Discrete Algorithms (ACDA21)*, pages 88–99. SIAM, 2021.

[14] Sidney Wayne Hess, JB Weaver, HJ Siegfeldt, JN Whelan, and PA Zitlau. Nonpartisan political redistricting by computer. *Operations Research*, 13(6):998–1006, 1965.

[15] Hisao Ishibuchi, Ryo Imada, Yu Setoguchi, and Yusuke Nojima. How to specify a reference point in hypervolume calculation for fair performance comparison. *Evolutionary computation*, 26(3):411–440, 2018.

[16] Aaron R Kaufman, Gary King, and Mayya Komisarchik. How to measure legislative district compactness if you only know it when you see it. *American Journal of Political Science*, 65(3):533–550, 2021.

[17] Myung Jin Kim. Multiobjective spanning tree based optimization model to political redistricting. *Spatial Information Research*, 26(3):317–325, 2018.

[18] Scott Kirkpatrick, C. Daniel Gelatt, and Mario P. Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[19] Harry A Levin and Sorelle A Friedler. Automated congressional redistricting. *Journal of Experimental Algorithmics (JEA)*, 24:1–24, 2019.

[20] Yan Y Liu, Wendy K Tam Cho, and Shaowen Wang. Pear: a massively parallel evolutionary computation approach for political redistricting optimization and analysis. *Swarm and Evolutionary Computation*, 30:78–92, 2016.

[21] Michael D McDonald and Robin E Best. Unfair partisan gerrymanders in politics and law: A diagnostic applied to six cases. *Election Law Journal*, 14(4):312–330, 2015.

[22] National Institute of Standards and Technology. D-Optimal Designs. https://www.itl.nist.gov/div898/handbook/pri/section5/pri521.htm.

[23] Richard G Niemi, Bernard Grofman, Carl Carlucci, and Thomas Hofeller. Measuring compactness and the role of a compactness standard in a test for partisan and racial gerrymandering. *The Journal of Politics*, 52(4):1155–1181, 1990.

[24] Yaghout Nourani and Bjarne Andresen. A comparison of simulated annealing cooling strategies. *Journal of Physics A: Mathematical and General*, 31(41):8373, 1998.

[25] Antonio GN Novaes, JE Souza de Cursi, Arinei CL da Silva, and João C Souza. Solving continuous location–districting problems with voronoi diagrams. *Computers & operations research*, 36(1):40–59, 2009.

[26] Daniel D Polsby and Robert D Popper. The third criterion: Compactness as a procedural safeguard against partisan gerrymandering. *Yale L. & Pol'y Rev.*, 9:301, 1991.

[27] Reynolds v. Sims. 377 U.S. 533 (1964).

[28] Federica Ricca and Bruno Simeone. Local search algorithms for political districting. *European Journal of Operational Research*, 189(3):1409–1426, 2008.

[29] EA Rincón-García, MA Gutiérrez-Andrade, SG De-los Cobos-Silva, P Lara-Velázquez, AS Ponsich, and RA Mora-Gutiérrez. A multiobjective algorithm for redistricting. *Journal of applied research and technology*, 11(3):324–330, 2013.

[30] Maral Shahmizad and Austin Buchanan. Political districting to minimize county splits. *Available on Optimization-Online*, 2023.

[31] Takeshi Shirabe. Districting modeling with exact contiguity constraints. *Environment and Planning B: Planning and Design*, 36(6):1053–1066, 2009.

[32] Nicholas O Stephanopoulos and Eric M McGhee. Partisan gerrymandering and the efficiency gap. *The University of Chicago Law Review*, pages 831–900, 2015.

[33] Lukas Svec, Sam Burden, and Aaron Dilley. Applying voronoi diagrams to the redistricting problem. *The UMAP journal*, 28(3):313–329, 2007.

[34] Dmitri I. Svergun. Restoring low resolution structure of biological macromolecules from solution scattering using simulated annealing. *Biophysical journal*, 76(6):2879–2886, 1999.

[35] Rahul Swamy, Douglas M King, and Sheldon H Jacobson. Multiobjective optimization for politically fair districting: A scalable multilevel approach. *Operations Research*, 2022.

[36] Leonardo Vanneschi, Roberto Henriques, and Mauro Castelli. Multi-objective genetic algorithm with variable neighbourhood search for the electoral redistricting problem. *Swarm and Evolutionary Computation*, 36:37–51, 2017.

[37] William Vickrey. On the prevention of gerrymandering. *Political Science Quarterly*, 76(1):105–110, 1961.

[38] Samuel S-H Wang. Three practical tests for gerrymandering: Application to maryland and wisconsin. *Election Law Journal*, 15(4):367–384, 2016.

[39] Wesberry v. Sanders. 376 U.S. 1 (1964).

[40] David Bruce Wilson. Generating random spanning trees more quickly than the cover time. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 296–303, 1996.