THE PENNSYLVANIA STATE UNIVERSITY
SCHREYER HONORS COLLEGE

SCHOOL OF SCIENCE, ENGINEERING, AND TECHNOLOGY

EVALUATING REDISTRICTING IN PENNSYLVANIA USING MONTE CARLO SIMULATION

NATHANIEL NETZNIK
Spring 2022

A thesis
submitted in partial fulfillment
of the requirements
for baccalaureate degrees
in Mathematical Sciences and Computer Science
with honors in Mathematical Sciences

Reviewed and approved* by the following:

J. Brian Adams
Associate Teaching Professor of Mathematical Sciences
Thesis Supervisor

Jeremy Blum
Associate Professor of Computer Science
Faculty Reader

Ronald Walker
Associate Professor of Mathematical Sciences
Honors Adviser

* Electronic approvals are on file.

ABSTRACT

For over 200 years, the integrity of elections in the United States has been threatened by the practice of partisan gerrymandering – drawing an electoral district map in such a way that one political party is favored over another. Opponents have spent several decades challenging allegedly gerrymandered district maps in both state and federal courts. These attempts have been largely unsuccessful. In these cases, the courts have commonly declared that they desire a more rigorous standard by which district maps can be evaluated before overruling them due to suspected gerrymandering. Researchers have developed mathematical metrics, statistical tests, and computer simulations for achieving a possible standard.

We sought to build upon this research by developing an alternative computer simulation. Expanding upon our previous research, we developed a model for evaluating the fairness of a district map given its real-world outcome. Provided geographic and voter data, our model generates a sequence of randomly drawn district maps by joining counties and precincts into a given number of districts that align reasonably with certain federal and Pennsylvania state-level requirements. The model then determines the number of Democratic and Republican districts in each map. The model compiles these counts into probability distribution for the number of Democratic and Republican districts in a randomly drawn map. This model, provided appropriate data, can be used to evaluate a real-world election result.

Our model determined that the 2016 House of Representatives and 2016 State House elections were fair. However, we identified potential limitations that should be addressed in a future model.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank my thesis supervisor, Dr. Adams, for introducing me to the topic of gerrymandering and for his continued support since we began our research on this topic in Fall of 2020;

my faculty reader, Dr. Blum, for the time he has put into reviewing my research;

my academic and honors adviser, Dr. Walker, for his outstanding support throughout my time in the honors program and as a student at Penn State Harrisburg;

the faculty and staff of Penn State Harrisburg's Mathematics and Computer Science department for teaching me much of what I know in mathematics, statistics, and computer science;

our dataset's GitHub contributors for their efforts to provide open, precinct-level election data;

my family and friends for their unconditional support throughout my time as a Penn State student and as a Schreyer Scholar.

Chapter 1

Introduction

The United States Constitution requires that states be divided into electoral districts following the decennial census. However, it offers surprisingly little guidance on how each state's districts are to be drawn, simply requiring that they be approximately equal in population (U.S. Const. art. I, § 2). It is left up to individual states to determine how districts should be drawn. In Pennsylvania, for example, congressional districts are drawn by a committee consisting of five members: two from the majority party, two from the minority party, and one chairman selected by these four. If these four cannot agree on the fifth member, the fifth is appointed by the state Supreme Court. The committee proposes a district map for the commonwealth, made official with the governor's signature (PA Const. art. II, § 17).

A fault of the decennial redistricting process is gerrymandering: the practice of dividing or arranging voting districts in such a way that favors one political party or candidate (Merriam-Webster, 2021). For over 200 years, the United States has used this for partisan gain. The practice is named after Elbridge Gerry, Governor of Massachusetts in 1812, whose party proposed a redistricting plan with a district that some said resembled a salamander (Cox & Katz, p. 3). Since that time, partisan gerrymandering has become a common practice (Kang, 2020). Gerrymandering is done through "cracking" – spreading the opposition party's voters out across many districts such that they fall just short of victory – and "packing" – concentrating the other party's voters into a small number of districts (McGhee & Stephanopoulos, 2015). Though the

practice has existed for over 200 years, modern computers allow for it to be done more effectively than ever before (Kang, 2020).

Despite the bipartisan structure of Pennsylvania's committee, gerrymandering still occurs; the infamous "Goofy kicking Donald Duck" district is a striking example (Criss, 2019).

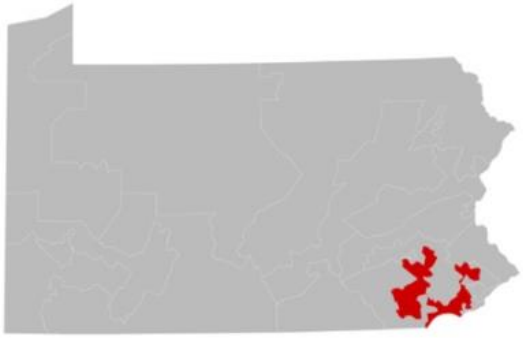

Figure 1: the "Goofy kicking Donald Duck" district (Criss, 2019)

The effect of gerrymandering is resulting in concerns over the fairness of elections. The United States has long espoused the ideal of "one man, one vote". (*Reynolds et al. Sims et al.*, 1964). Court cases have been filed alleging that district maps have been drawn unfairly; until recently, few plaintiffs have been successful. The challenge in these cases has been the lack of a manageable standard by which to quantify the impact of gerrymandering, and the lack of a corresponding threshold at which a partisan gerrymander has changed what should have been the result of a fair election. In recent years, researchers have attempted to address these concerns by creating summary metrics, geometric measures, and computer simulations by which district plans can be evaluated.

Chapter 2

Justiciability of Partisan Gerrymandering

A significant challenge of addressing partisan gerrymandering is justiciability - defining a measure for its severity that will be accepted by the courts. While *racial* gerrymandering has been ruled illegal (*Miller et al. v. Johnson et al.*, 1995; Voting Rights Act of 1965, 1965), partisan gerrymandering has proven to be a more complicated matter. In recent years, several cases have been brought before courts regarding alleged partisan gerrymanders. Nevertheless, courts have failed to resolve two critical questions: by what standard can a court of law measure the fairness of a district map, and at what point has the partisan gerrymander gone too far?

In 1986, the Supreme Court declared that partisan gerrymandering is justiciable (*Davis et al. v. Bandemer et al.,* 1986). At the federal court level, plaintiffs have since brought forth a variety of reasons by which they believed partisan gerrymanders to be unlawful including the Equal Protection Clause of the 14th amendment, the *one person, one vote* principle, and the 1st amendment. Each of these reasons has been struck down (*Rucho et al. v. Common Cause et al.*, 2019).

Supreme Court Justices from both ends of the political spectrum have expressed interest in establishing a metric for quantifying partisan gerrymandering (*League of United Latin American Citizens et al. v. Perry et al.*, 2006; *Vieth et al. v. Jubelirer et al*., 2004). In spite of this, opponents to gerrymandering were struck down again in 2019 when the Supreme Court ruled that it could not intervene in such cases, citing a lack of "judicially discoverable and manageable standards for resolving [them]" (*Rucho et al. v. Common Cause et al.*, 2019). The Supreme Court has, however, pointed out that partisan gerrymandering can be addressed through state amendments or through the Congress (*Rucho et al. v. Common Cause et al.,* 2019).

Initiatives in this realm, including those decided by District Courts, have seen greater success (*Common Cause et al. v. Lewis et al.*, 2019; *Whitford et al. v. Gill et al.*, 2016; *League of Women Voters of Pennsylvania et al. v. the Commonwealth of Pennsylvania*, 2018; *Rucho et al. v. Common Cause et al.*, 2019).

There have been several efforts to develop measures and methodologies for quantifying gerrymandering in a court setting (Chen, 2017; Herschlag et al., 2020; McGhee & Stephanopoulos, 2015; Wang, 2016). Some of these measures have seen success. McGhee and Stephanopoulos's efficiency gap helped plaintiffs win a case in which the District Court for the Western District of Wisconsin ordered that gerrymandered districts be redrawn (*Whitford et al. v. Gill et al.*, 2016). Chen's research (2017), driven by computer simulation, played a role in a Court decision to strike down gerrymandered maps in North Carolina (*Common Cause et al. v. Lewis et al*., 2019).

Whether or not partisan gerrymandering can be addressed by federal courts, there is still the possibility for standards to be put in place at the state level or through congress. This yields a need for continued research in improving existing metrics for partisan gerrymandering, or for creating new metrics.

Chapter 3

A Review of Quantifying Gerrymandering

Summary Metrics

Many researchers have attempted to quantify partisan gerrymandering by proposing computationally simple summary metrics. Some metrics are heavily rooted in statistical theory, while others have simply quantified notions that Supreme Court Justices have considered with respect to gerrymandering. With their simplicity and the ability to define them based on legally sound principles, summary metrics have apparent potential in a court setting. One of the most notable of these metrics, the efficiency gap, has been considered and applied in court (*Whitford et al. v. Gill et al.*, 2016).

Partisan symmetry is the notion that a given number of votes for either party should translate directly to a certain number of legislative seats, regardless of which party has what percentage of the votes (McGhee & Stephanopoulos, 2015). For example, suppose Party A wins 60% of the vote in a state and is given 10 seats. If partisan symmetry holds, then in a hypothetical election where Party B wins 60% of the vote, they should also be given 10 seats – no more, and no less. This is considered a meaningful metric with respect to gerrymandering, as cracking and packing would be expected to skew this symmetry.

Discussions between Supreme Court Justices on using partisan symmetry as a means for measuring partisan gerrymandering inspired McGhee and Stephanopoulos to propose their efficiency gap metric. The efficiency gap quantifies partisan symmetry in terms of "wasted votes" - that is, the number of votes that are either cast for a losing candidate or for the winning candidate in excess of what is required to win. The efficiency gap is defined as the difference

between each parties' wasted votes divided by the total number of votes cast (McGhee & Stephanopoulos, 2015). This measure has played a significant role in a court setting (*Whitford et al. v. Gill et al.*, 2016). Other researchers have since studied various mathematical properties of this metric, proposed variations, or proposed algorithms for minimizing its value (Chatterjee et al., 2020; Tapp, 2019).

Other metrics have been proposed as well. Following a partisan gerrymander, one might expect that the party favored by the gerrymander will have a large number of marginal victories, while the party targeted by the gerrymander will have a few landslide victories. One approach to measuring this is to consider the difference between the share of Democratic votes in districts that Democrats win and the share of Republican votes in districts that Republicans win (Wang, 2015). Another approach is to use a mean-median difference: the difference between a party's mean and median share of votes among districts (Wang, 2015). If either of these differences are far from zero, this could indicate that a party has been "packed" heavily into a few outlier districts.

Though summary metrics are often convenient, they do have pitfalls. While the efficiency gap has been championed for playing a role in *Whitford v. Gill*, the case sparked discussion on some of its concerns. First, the constitution does not require that votes translate proportionally into seats - a notion that the efficiency gap assumes. Moreover, some speculated that the efficiency gap could be skewed by the political geography of a state (*Whitford et al. v. Gill et al., 2016*). A state may have a natural underlying political bias if it contains, for example, a predominantly Democratic urban city or predominantly Republican rural grounds (Chen & Rodden, 2013). None of the other metrics discussed in this section account for geopolitical makeup either, as they are simply calculated from a given plan without greater context.

Researchers have proposed metrics which address this (Herschlag, 2020), but such metrics are sparse and not well studied.

Geometric Compactness

From the time of Elbridge Gerry's "salamander", accusers have pointed to bizarre district boundaries as evidence of gerrymandering (Alexeev & Mixon, 2018). In a court case on an alleged racial gerrymander, Supreme Court Justices claimed that bizarrely shaped districts warrant scrutiny (*Shaw v. Reno,* 1993). Measures of compactness, those which describe the "regularity" of a shape, can be applied these unusual districts. In fact, some states require that their districts be compact (Kaufman et al., 2020). Various categories of compactness measures have been applied to evaluate redistricting plans. Some of these measures are general-purpose, while others were designed particularly for the domain of gerrymandering.

Since the early 1800s, mathematicians have proposed measures to quantify compactness. Historically common approaches to doing this include ratios between perimeter and area, comparison with standard shapes such as circles and squares, and the disbursement of area away from the shape's center (Maceachren, 1986). Such measures can be applied in attempts to detect partisan gerrymanders. Some researchers have devised metrics particularly for measuring gerrymandering to take into account problem-specific factors such as the geopolitical distribution of voters (Fryer & Holden, 2007).

Another measure of compactness is convexity; a shape is convex if it contains the shortest path between any two points within. Many simple shapes such as circles, squares and triangles are convex, while irregular shapes tend to be non-convex (Chambers & Miller, 2010).

In more recent years, researchers have measured district convexity by estimating the probability that the line between a given pair of points is contained within the district (Chambers & Miller, 2010; Hodge et al., 2010). Such probabilities can be estimated using computer simulation (Hodge et al., 2010).

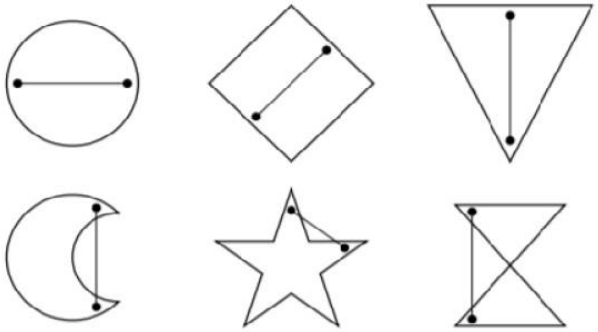

Figure 2: examples of convex (top) and non-convex (bottom) shapes (Chambers & Miller, 2010)

Despite many states requiring their districts be compact, some researchers have questioned whether geometric compactness alone is a sufficient measure for detecting gerrymandering (Alexeev & Mixon, 2018). Regardless, the issue of manageable standards persists, as these states have not yet agreed on how to precisely define and measure compactness. Additionally, requiring districts to be compact might disadvantage certain subgroups of voters, such as Democratic voters in a city (*Vieth et al. v. Jubelirer et al.*, 2004). Once again, few of these measures account for the geopolitical makeup of a region. One might also consider how some states are naturally non-compact, such as the state of Maryland. Detecting gerrymandering by compactness could be more difficult in such states.

Computer Simulation

In light of modern-day computing technologies, it has become common for researchers to use computer simulations to study gerrymandering. These simulations generally involve generating a large set of possible redistricting plans, observing their properties, and comparing a proposed plan to those in the set to determine if it is an outlier with respect to the others. A major benefit of using computer simulation versus other metrics in studying gerrymandering is that it can account for the geopolitical makeup of voters within a state. By combining historical voting data with computer generated redistricting plans, one can study hypothetical election results relative to a state's distribution of voters (Adams & Netznik, 2021).

Many researchers have used a class of sampling methods known as Markov chain Monte Carlo (MCMC) algorithms to generate sets of plans (Barkstrom et al, 2019; Fifield et al., 2020; Herschlag et al, 2020). Other algorithms generate plans by selecting random units of a region as starting points for districts, connecting adjacent units in such a way that desirable districts are constructed (Adams & Netznik, 2021; Chen & Cottrell, 2016; Cirincione et al., 2000). One must be cautious that their simulation is efficient, as tasks such as generating sets of district maps can be computationally intensive (Fifield, 2020; Barkstrom et al., 2019). Chen's computer simulation played a role in striking down gerrymandered maps in a 2019 court case (Chen, 2017; *Common Cause et al. v. Lewis et al*., 2019).

Summary

The last few decades have exhibited a rising concern over the topic of gerrymandering. Following the Supreme Court's historic *Rucho v. Common Cause* ruling in 2019, along with the

continued need for unambiguous and complete measures by which to detect and quantify gerrymanders, addressing it has perhaps never been more urgent. But with technology advancing, mathematics maturing, and awareness increasing, the issue has never been more addressable. Bringing together the resources that we have today, the systematic and academic study of gerrymandering is crucial for policymakers to write meaningful and effective regulations by which to protect the public from its consequences.

Prior work on gerrymandering brings about a variety of addressable issues. One of the most pressing of these concerns is the creation of a "manageable standard" by which to measure a gerrymander in a court setting. Future research could entail creating new metrics, or improving others' metrics, for doing so. Summary metrics could be extended to take a wider variety of factors into account. There is also potential for further investigation into geometric compactness as it relates to gerrymandering. In the realm of computing, one could work towards creating more efficient and more realistic simulations.

Chapter 4

Methodology

We propose a Monte Carlo simulation that can be used to evaluate the results of an election. In our previous work (Adams & Netznik, 2021) we devised a simple simulation that could randomly generate district maps by connecting adjacent precincts into districts, then create a probability distribution for the outcome of an election upon calculating the number of Democratic majority and Republican majority districts in each map. Building off this, we will revise our simulation to better account for significant redistricting constraints – namely, that districts have approximately equal populations, are contiguous, and avoid breaking county lines unless necessary. We will then run our simulation on a set of real-world data. By using the results of 2016 elections, it should be possible to determine what fair districts are, and how they compare to the actual – potentially gerrymandered – districts.

Original Monte Carlo Simulation

Our goal is to generate a set of randomly drawn district maps for a given region. The basic approach that we will use to achieve this, established in our previous work, is a Monte Carlo simulation technique known as bootstrapping. Bootstrapping involves generating random values resulting from a process – in our case, the number of districts that each party wins based on a randomly drawn district map. Specifically, given the k^{th} n-district map in a sequence of N maps, the number of districts in the map won by each party is represented by the random variable

$$X_k = (x_{k1}, x_{k2})$$

where x_i represents the number of districts won by party i and $n = x_1 + x_2$. This results in a probability distribution with random variables

$$X_k = \{(0, n), (1, n-1), \ldots, (n, 0)\}.$$

The goal of our original simulation, laid out in our previous work, was to generate n redistricting plans (maps). We began by reading in sample data for a region's electoral precincts. We then modeled the precincts as nodes of an initially unconnected graph, and districts as trees consisting of connected precincts. To construct the first district, we randomly selected an initial precinct as a starting point. We then randomly connected unselected adjacent precincts to the district until the district reached a set population threshold. This construction process was repeated until N districts had been formed. Using the precinct data, we calculated the winning party (Republican or Democrat) of each district from each generated map by popular vote, storing the number of Republican and Democratic districts resulting from each map. This entire process was repeated n times, resulting in n randomly generated maps (Adams & Netznik, 2021).

This bootstrapping procedure provided a simple and practical way to construct the maps; however, before applying our simulation to real world data, we need to address a few pitfalls.

Improvements

Implementing Redistricting Constraints

In our original simulation, our only redistricting constraint is that each district is contiguous and that all districts have approximately equal populations. We will expand our

criteria to better account for senatorial and representative redistricting criteria established in the Pennsylvania state constitution: namely, that “unless absolutely necessary no county, city, incorporated town, borough, township or ward shall be divided” (PA Const. art. II, § 16). To reduce the amount that districts break municipal boundaries, we will attempt to add entire counties to districts before adding individual precincts.

Including All Precincts

One significant concern with our original simulation is that it does not guarantee that all precincts will be connected to form the desired number of contiguous districts. It is likely that partially constructed districts and groups of unchosen precincts get “landlocked” by surrounding districts.

In this model, as the simulation runs, if it finds that a district has no more unchosen adjacent precincts before reaching the population threshold, the district’s construction process is terminated. This resulted in districts with insufficient populations as well as precincts that are not appended to any district.

An approach that we considered for addressing this issue is as follows: once all district construction has been terminated, the model begins to cycle through all remaining unchosen precincts. If an unchosen precinct is adjacent to a district, it is added to the district. This process is repeated until all precincts have been added to one of the districts.

A problem with this approach is that some districts will likely have significantly higher populations than others. To alleviate this, the model begins swapping precincts from

overpopulated districts to less populated districts. This continues until the district populations are sufficiently balanced.

This approach is not perfect. It may occur that a district is split by the swapping process. Hypothetically, one district could swap a precinct to an adjacent district, followed by a sequence of swaps that loops back into the original district, cutting off a portion of the underpopulated district. For the swapping process to yield constitutional, contiguous districts, the simulation would need a way to detect when continuity is broken, or to anticipate when it will be.

Another approach we considered was to repeatedly generate maps, only keeping those that include all, or nearly all, of the region's precincts. We ascertained that it would be infeasible to generate a fully constructed map without precincts and partially completed districts being landlocked by completed districts. Therefore, we chose to use approach but only keeping maps with a desired number of fully constructed districts.

To obtain results using this approach, we will begin by using the repeated map generation approach to generate and keep partial maps with nearly all of its districts fully constructed. For example, if we are modeling the state of Pennsylvania and desire 18 districts, we could construct maps with at least 15 fully constructed and no more than 3 partially constructed districts. We will then cycle through each of the unchosen precincts and add them to the partially constructed districts. While the final districts will not all be contiguous, the frequency table generated by these maps should provide acceptable results.

Data

Our data is part of a dataset from https://github.com/nvkelso/election-geodata, a repository containing precinct-level geographic data and vote results for several states. The data in the repository has been compiled from a variety of sources including election officials, journalists, and the 2010 Census. The particular data for Pennsylvania that we used was obtained from https://github.com/nvkelso/election-geodata/tree/master/data/42-pennsylvania/statewide/2017. This dataset contains a shapefile representing Pennsylvania's 9152 precincts with data from the mid-2010s. The relevant data that we will be using includes shape data – coordinates for each of Pennsylvania's precincts – and records for each of Pennsylvania's precincts including county name, population, and number of voters by party for several elections.

Revised Simulation

In this section we will outline each aspect of our revised simulation in further detail. Our simulation is written in Python; we edited and ran it using PyCharm.

Object Structures

We created three Python classes to store data for each of the geographic units accounted for in our simulation – districts, counties, and precincts.

We first instantiate the precinct objects. We assign IDs sequentially. We then obtain precinct population, county name, Democratic votes, and Republican votes from the shapefile records, where each record pertains to a precinct. Each precinct is stored in a list.

Before instantiating the county objects, we collect a list of counties by cycling through the shapefile records, collecting each unique county name. To instantiate the county objects, we again assign IDs sequentially. We obtain the county name from the list of counties we created previously. Each county object also contains a list of its precincts, obtained by scanning the list of precinct objects for all precincts that share the county name. County populations, Democratic votes, and Republican votes are computed by summing the respective fields from each of the precincts within the county.

We instantiate districts with a $phase$ value of 0; the phase value is 0 if the simulation is adding counties to the district, and 1 if adding individual precincts. Each district is also given an empty list of precincts and empty sets of choices for the next county or precinct to append. The district population, Democratic vote count, and Republican vote count are each set to 0. Each district is also given an empty color field (this will be used if plotting districts). Finally, each district keeps track of an initially empty current county – this will be relevant in the initial district construction process.

Data Preprocessing

We preprocessed the data to generate each adjacency list used in the district construction process. We generated three adjacency lists – precinct-precinct, county-precinct, and county-county – each mapping the former units to their adjacent latter units. These lists will be used in determining which units can be added next to a district in construction.

To create the precinct-precinct adjacency list, we obtained precinct latitude-longitude coordinates from the shapefile and rounded each to five decimal places to eliminate any existing

rounding error. We then stored each precinct's coordinates in a set and used set intersections to determine which precincts contain overlapping coordinates. To construct the county-precinct and county-county adjacency lists we cycled through each precinct with its respective precinct adjacency list, noting the county associated with each precinct. We used this information to construct the lists. The adjacency lists were written to text files, which are read each time the simulation is run.

Initial District Construction

Once the data is preprocessed, we enter the main method of the simulation where district construction occurs. The main method takes in five parameters:

- m: population of the entire region
- n: desired number of districts
- num_runs: the number of complete district maps to be generated
- $complete\_threshold$: the minimum population desired for a district – any district with a smaller population will be considered incomplete
- $min\_complete\_count$: the minimum number of completed districts for a complete map

The majority of district construction takes place in an initial phase where we apply two methods: $initialize()$ and $add\_unit()$. To begin the district construction process, we randomly select an unchosen precinct and attempt to $initialize()$ a district with the precinct's county; if the county's population exceeds the threshold of m/n, rather than calling $initialize()$ with the county, we advance to phase 1 and $initialize()$ with the precinct. Once each of the n districts has been initialized, we continue to cycle through each of them, applying $add\_unit()$ district-by-

district. The simulation terminates a district's construction process when either it's population exceeds the threshold or it fails to identify adjacent unchosen precincts.

Provided a county (phase 0) or precinct (phase 1), the $initialize()$ method appends each of the provided unit's precincts to the district's precinct list, updating the lists of unchosen precincts and counties accordingly. It then updates the district's list of possible next choices for adding a precinct (and adding a county, if in phase 0) by pulling the appropriate elements from the precinct-precinct and county-precinct adjacency lists, ensuring to only add those elements that have not been appended to any district. If in phase 1, the simulation will update the district's $current\_county$ field. The district's population, Democratic votes, and Republican votes are then updated accordingly.

Once all districts have been initialized, the simulation begins to apply the $add\_unit()$ method. Provided a district, this method will add an adjacent precinct or county depending on the district's phase.

In phase 0, the method determines if there are any adjacent counties that can be added to the district. If there are none, the district advances to phase 1 and the method returns. Otherwise, the method will choose a random adjacent county to append to the district. If the district's population plus the total population of all unchosen precincts in the county exceeds the district's population threshold, the district advances to phase 1 (see below). Otherwise, the county is added to the district and the district's fields are updated accordingly.

In phase 1, the method determines if there are any unchosen adjacent precincts that can be added to the district. If there are none, the district's initial construction process terminates. Otherwise, the simulation randomly chooses an adjacent precinct to add. The simulation will first try to choose an adjacent precinct from $current\_county$ - the county that either going into

$initialize()$ or in a previous application of $add\_unit()$ could not be appended. If no such precinct is available, it will choose a precinct from outside of $current\_county$. If adding this precinct would send the district's population beyond m/n, the district's initial construction process is terminated; otherwise the precinct is added and the district's fields are updated accordingly.

Appending Unchosen Precincts

At this point, each district has either reached its population threshold or has no adjacent units to append. To generate a complete map, we must cycle through any remaining unchosen precincts and append them to districts.

The simulation counts the number of districts that have been populated at least to the provided minimum district population $complete\_threshold$. If this count falls short of the minimum number of complete districts desired $min\_complete\_count$, the initial map is discarded and the run is restarted. If all districts have reached the minimum district population, each of the remaining unchosen precincts is appended to the district with the smallest initial population. Otherwise, the simulation cycles through each of the initially underpopulated districts, adding a remaining unchosen precinct each time.

Creating the Probability Distribution

Once all precincts have been appended to districts, the map is complete and we can use voter data to compute the winning party of each district. To do this, we begin by count the total votes for each party by districts then tallying the number of predominantly Democratic and

predominantly Republican districts in the map. We then update a list, d_wins, that stores the number of times in the sequence of runs that the Democratic party wins i of n districts at index i. At this point, we repeat the simulation until we have constructed the desired number of complete district maps.

Once all runs are complete, we construct and print a table for the probability distribution using the d_wins list. The first two columns correspond to the number of Democratic and Republican districts in a map, respectively. The third column reports the number of times that each of those outcomes occurred in the simulation, and the fourth column divides that count by the number of maps to obtain a probability.

Table 1: Structure of probability distribution

D	R	F	P
0	n	$d\_wins[0]$	$d\_wins[0]/n$
1	$n-1$	$d\_wins[1]$	$d\_wins[1]/n$
⋮	⋮	⋮	⋮
i	$n-i$	$d\_wins[i]$	$d\_wins[i]/n$
⋮	⋮	⋮	⋮
n	0	$d\_wins[n]$	$d\_wins[n]/n$

Plotting Maps

Finally, the simulation can plot two types of maps for a run – one being a plot of each unique district, and one being a red-blue map showing the geopolitical distribution of a map (Republican districts are plotted red and Democratic districts are plotted blue). Using Python's matplotlib library, the simulation loops through a list of districts, plotting each as a color – a

randomly chosen color from a list if plotting the unique districts, and red or blue if plotting the geopolitical distribution.

Chapter 5

Results

2016 U.S. House of Representatives Election

We simulated Pennsylvania's 2016 U.S. House of Representatives election. We used total population $m = 12732284$, number of districts $n = 18$, number of runs $num\_rums = 100$, minimum initial district population $complete\_threshold = 670000$ (roughly 95% of m/n), and minimum complete district count $min\_complete\_count = 15$. We also plotted a district map and a red-blue map. The results are presented below.

Table 2: Results of simulation for 2016 U.S. House of Representatives Election

D	R	F	P
0	18	0	0
1	17	0	0
2	16	0	0
3	15	3	0.03
4	14	21	0.21
5	13	42	0.42
6	12	28	0.28
7	11	6	0.06
8	10	0	0
⋮	⋮	⋮	⋮

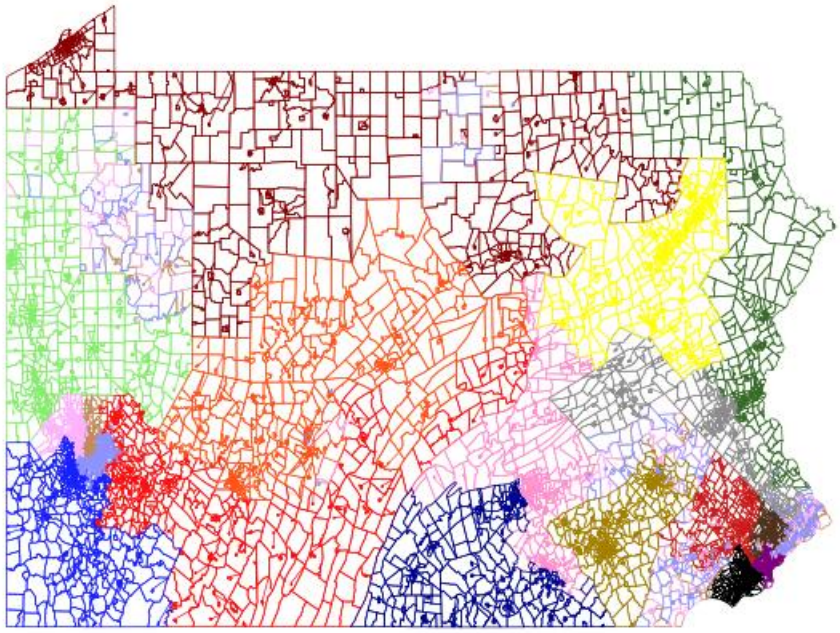

Figure 3: Sample district plot for 2016 U.S. House of Representatives

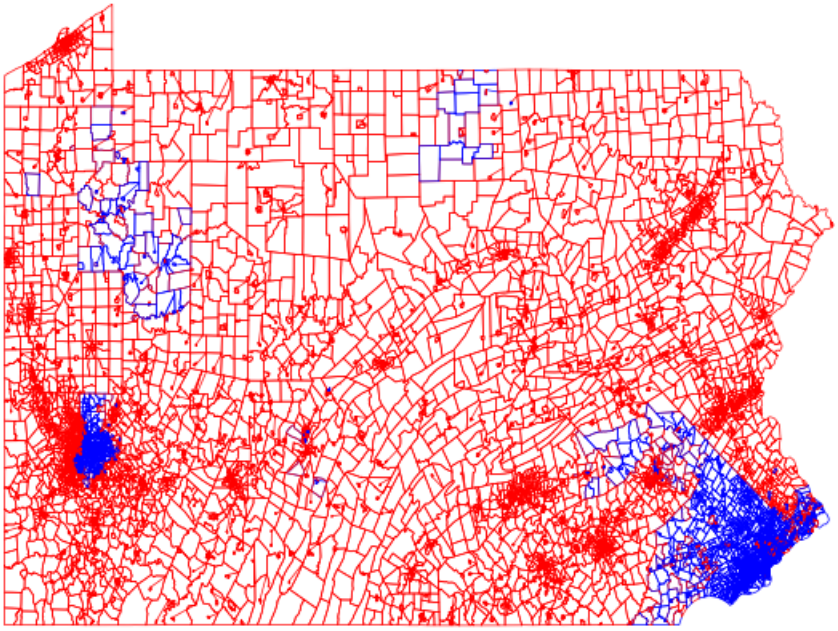

Figure 4: Sample red-blue plot for 2016 U.S. House of Representatives

2016 State House Election

We simulated Pennsylvania's 2016 State House election. We used total population $m = 12732284$, number of districts $n = 203$, $num\_runs = 100$, minimum initial district population $complete\_threshold = 40000$, and minimum complete district count $min\_complete\_count = 160$. We also plotted a district map and a red-blue map. The results are presented below.

Table 3: Results of simulation for 2016 State House Election

D	R	F	P
0	203	0	0
⋮	⋮	⋮	⋮
74	129	0	0
75	128	1	0.01
76	127	0	0
77	126	3	0.03
78	125	2	0.02
79	124	3	0.03
80	123	9	0.09
81	122	7	0.07
82	121	6	0.06
83	120	16	0.16
84	119	8	0.08
85	118	4	0.04
86	117	13	0.13
87	116	5	0.05
88	115	7	0.07
89	114	5	0.05
90	113	4	0.04
91	112	1	0.01
92	111	3	0.03
93	110	1	0.01
94	109	2	0.02
95	108	0	0
⋮	⋮	⋮	⋮

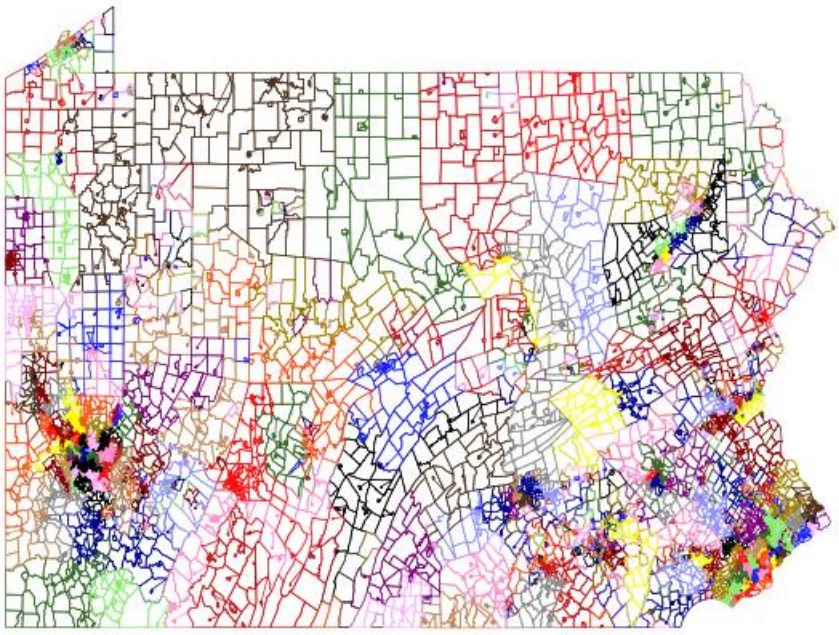

Figure 5: Sample district plot for 2016 State Senate

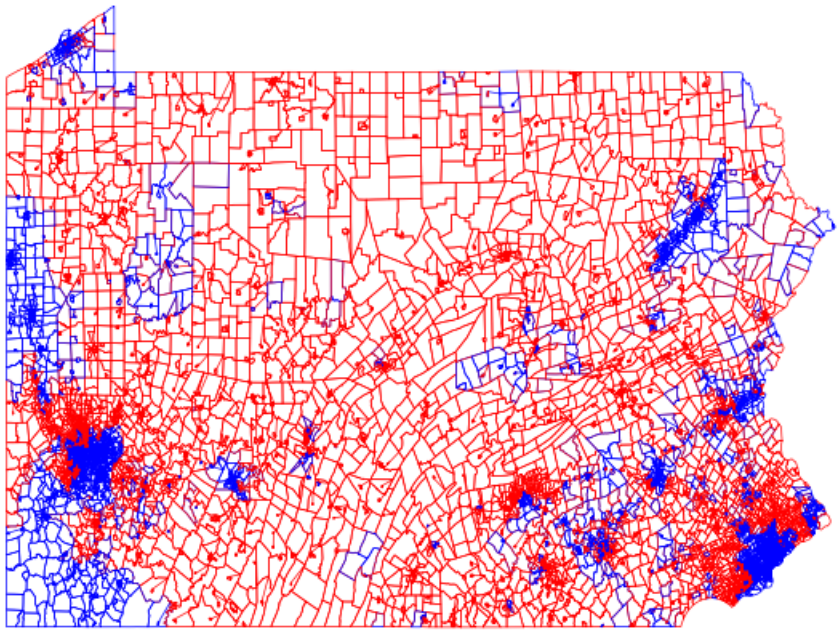

Figure 6: Sample red-blue plot for 2016 State Senate

Interpretation of Results

The 2016 U.S. House of Representatives election resulted in 5 Democratic and 13 Republican seats (2016 Pennsylvania House Election Results, n.d.). With a 0.05 level of significance, we can expect a fair district map to contain between (3D, 15R) and (7D, 11R) districts, inclusive. The 2016 U.S. House of Representatives election was fair according to our results.

The 2016 State House election resulted in 82 Democratic and 121 Republican seats (Pennsylvania House of Representatives elections, 2016, n.d.). With a 0.05 level of significance, we can expect a fair district map to contain between (77D, 126R) and (93D, 110R) districts inclusive. Pennsylvania's 2016 State Senate election was fair according to our results.

Limitations

Note that these conclusions assume that our model is generating a representative sample of possible district maps. Due to Pennsylvania's redistricting constraints that we enforced and various decisions we made regarding how our algorithm generates maps, the possible maps generated by our model are limited. Additionally, these conclusions assume that the probability distribution generated by the sample of maps is the same as the probability distribution for all possible maps.

We note that the simulated results for the 2016 U.S. House of Representatives lean heavily in favor of the Republican party. There are two possible theories for this. The first theory is that this results from a pitfall in our simulation. Because our simulation attempts to add counties before individual precincts, and Republican-leaning counties have smaller populations

than Democratic-leaning counties, Republican districts are constructed more quickly and landlock Democratic districts. These Democratic districts terminate early, and surrounding Democratic voters end up packed into surrounding Republican districts. We believe that this theory is a probable explanation for our results.

An alternative theory, which could hold true in addition to the first, is that the large Republican district count results from a natural "packing" affect that arises from Pennsylvania's geopolitical makeup in conjunction with the nature of districting. When one follows the PA Constitution redistricting constraints of not crossing county and municipal lines unless necessary, it is inevitable that some counties and municipalities are too populated to fit entirely into one district. These highly populated counties and municipalities tend to be highly Democratic. Thus, they are split between districts, and Democratic voters are packed into Republican districts. This is expected to occur in any constitutional district map, even one that is not intentionally gerrymandered.

Chapter 6

Conclusions

Politicians and academic researchers alike have spent several years considering how to define a manageable standard for evaluating alleged partisan gerrymanders. We addressed this matter by designing and implementing a computer simulation that randomly generates a set of district maps that conform reasonably to several state and federal guidelines, counts the number of Democratic majority and Republican majority districts in each map, and complies these counts into a probability distribution by which a real-world election result can be evaluated against. Our model showed that both Pennsylvania's 2016 U.S. House of Representatives election and Pennsylvania's 2016 State House election were fair, and thus their associated district maps were not gerrymandered.

However, our model does have a few limitations. First, though we assume that the district maps our model generates represent the population of all possible maps, the possible maps generated by our simulation are constrained. We also assume that the probability distribution generated by these maps represents the probability distribution generated by all possible maps. In addition, some districts get landlocked before completion, resulting in non-contiguous districts and the possible packing of Democratic voters into surrounding Republican counties.

Future research could further address the limitation of districts being landlocked before completion. A particularly promising direction would be constructing districts using a "top-down" approach. Beginning with an entire state, use $initialize()$ and $add\_unit()$ to divide the state into two or three districts. We have found that it is not difficult to generate fully populated, contiguous maps with so few districts. From here, apply $initialize()$ and $add\_unit()$ to each of these subdivisions, dividing each of them into another two or three districts. If one chooses an

appropriate number of subdivisions at each step, they could continue this process until they have constructed the precise number of districts that they desire. For example, if generating 18 congressional districts, the state of Pennsylvania could be divided into two districts, each of those into three, and each of those into three. For a desired number of districts that cannot be factored into such small numbers, simply generate scattered initial districts until the remaining number of districts can be broken down. For example, if one desires 19 districts, one can create one district as presented in our methodology, then divide the remainder of the state into the remaining 18 districts by the process described above. This process would allow for the construction of complete district maps with exclusively contiguous districts. This would also address our theory that voters surrounding landlocked districts get packed into surrounding districts controlled by their opposing party.

Another direction could be to apply the "swapping" approach for appending remaining precincts, exploring graph-theoretic techniques for detecting when a swap would break continuity. One could also use demographic data to investigate racial gerrymandering. One could also apply our simulation in conjunction with other existing measures, such as the efficiency gap or measures of compactness.

Appendix A

Code for Preprocessing Data

```
import shapefile

class County:
    def __init__(self):
        self.id = -1
        self.name = ""
        self.precincts = []
        self.population = 0
        self.d_votes = self.r_votes = 0

class Precinct:
    def __init__(self):
        self.id = -1
        self.population = 0
        self.county_name = ""
        self.district = None
        self.d_votes = self.r_votes = 0

sf = shapefile.Reader("C:/Users/nhnet/Documents/College/Honors Thesis/Data Sources/VTDs_Oct17 (Precinct Shapefiles)/VTDs_Oct17.shp")
shapes = sf.shapes()
records = sf.records()

p_adjacency = {i: [] for i in range(len(shapes))}

# Round points to 5 decimal places
p = [[] for i in range(len(shapes))]
for i in range(len(shapes)):
    for j in range(len(shapes[i].points)):
        p[i].append(tuple([round(shapes[i].points[j][0], 5), round(shapes[i].points[j][1], 5)]))

# Store points as sets and use set intersections to determine if two districts
# are adjacent
points = [set(p[i]) for i in range(len(p))]

county_list = set()
for i in range(len(records)):
    county_list.add(records[i][23])
county_list = list(county_list)

county_list.sort()

# Initialize precinct and county objects
precincts = [Precinct() for i in range(len(records))]
counties = [County() for county in county_list]

for i in range(len(precincts)):
    precincts[i].id = i
    precincts[i].population = records[i][4]
    precincts[i].county_name = records[i][23]
    precincts[i].d_votes = records[i][39]
    precincts[i].r_votes = records[i][40]

for i in range(len(county_list)):
    counties[i].id = i
    counties[i].name = county_list[i]
    counties[i].precincts = [p for p in precincts if records[p.id][23] == counties[i].name]
    counties[i].population = sum([p.population for p in counties[i].precincts])
    counties[i].d_votes = sum([p.d_votes for p in counties[i].precincts])
    counties[i].r_votes = sum([p.r_votes for p in counties[i].precincts])
```

```
for i in range(len(shapes)):
    for j in range(i + 1, len(shapes)):
        if points[i].intersection(points[j]) != set():
            p_adjacency[i].append(j)
            p_adjacency[j].append(i)

#https://stackoverflow.com/questions/36965507/writing-a-dictionary-to-a-text-file
with open('precincts.txt', 'w') as f:
    print(precincts, file=f)

with open('counties.txt', 'w') as f:
    print(counties, file=f)

with open('p_adjacency.txt', 'w') as f:
    print(p_adjacency, file=f)

county_ind = {c.name: c.id for c in counties}

with open('county_ind.txt', 'w') as f:
    print(county_ind, file=f)

c_c_adjacency = {c.id: set() for c in counties}
c_p_adjacency = {c.id: set() for c in counties}
for p1 in p_adjacency:
    for p2 in p_adjacency[p1]:
        if p1 != p2:
            if records[p1][23] != records[p2][23]:
                c_c_adjacency[county_ind[records[p1][23]]].add(county_ind[records[p2][23]])
                c_p_adjacency[county_ind[records[p1][23]]].add(p2)

for c in counties:
    c_c_adjacency[c.id] = list(c_c_adjacency[c.id])
    c_p_adjacency[c.id] = list(c_p_adjacency[c.id])

with open('c_c_adjacency.txt', 'w') as f:
    print(c_c_adjacency, file=f)

with open('c_p_adjacency.txt', 'w') as f:
    print(c_p_adjacency, file=f)

if __name__ == "__main__":
    print()
```

Appendix B

Code for U.S. House of Representatives Election Simulation

```
"""
Goal: Create a Monte Carlo simulation to generate a discrete probability distribution for
the number of Democrat and Republican districts in an election with a randomly drawn map
"""

# Source used for syntax:: https://docs.python.org/3/

import ast
import matplotlib.pyplot as plt
import random as rand
import shapefile

class District:
    def __init__(self):
        self.complete = False
        self.phase = 0       # 0 if adding counties, 1 if adding precincts
        self.precincts = []
        self.c_choices = set()
        self.p_choices = set()
        self.population = 0
        self.d_votes = 0
        self.r_votes = 0
        self.color = ""
        self.current_county = County()

class County:
    def __init__(self):
        self.id = -1
        self.name = ""
        self.precincts = []
        self.population = 0
        self.d_votes = self.r_votes = 0

class Precinct:
    def __init__(self):
        self.id = -1
        self.population = 0
        self.county_name = ""
        self.district = None
        self.d_votes = self.r_votes = 0

n = 18

sf = shapefile.Reader("C:/Users/nhnet/Documents/College/Honors Thesis/Data Sources/VTDs_Oct17 (Precinct Shapefiles)/VTDs_Oct17.shp")
shapes = sf.shapes()
records = sf.records()

# Create initial list of counties
county_list = set()
for i in range(len(records)):
    county_list.add(records[i][23])
county_list = list(county_list)

county_list.sort()

# Initialize precinct and county objects
precincts = [Precinct() for i in range(len(records))]
counties = [County() for county in county_list]

for i in range(len(precincts)):
    precincts[i].id = i
```

```
    precincts[i].population = records[i][4]
    precincts[i].county_name = records[i][23]
    precincts[i].d_votes = records[i][39]
    precincts[i].r_votes = records[i][40]

for i in range(len(county_list)):
    counties[i].id = i
    counties[i].name = county_list[i]
    counties[i].precincts = [p for p in precincts if records[p.id][23] == counties[i].name]
    counties[i].population = sum([p.population for p in counties[i].precincts])
    counties[i].d_votes = sum([p.d_votes for p in counties[i].precincts])
    counties[i].r_votes = sum([p.r_votes for p in counties[i].precincts])

# Read in preprocessed  files
#https://www.kite.com/python/answers/how-to-read-a-dictionary-from-a-file-in--python

file = open("county_ind.txt", "r")
contents = file.read()
county_ind = ast.literal_eval(contents)
file.close()

file = open("c_c_adjacency.txt", "r")
contents = file.read()
c_c_adjacency = ast.literal_eval(contents)
file.close()

file = open("c_p_adjacency.txt", "r")
contents = file.read()
c_p_adjacency = ast.literal_eval(contents)
file.close()

file = open("p_adjacency.txt", "r")
contents = file.read()
p_adjacency = ast.literal_eval(contents)
file.close()

def main(m, n, num_runs, complete_threshold, min_complete_count):
    '''
    Conducts the simulation
        m: population size of region
        n: desired number of districts
        num_runs: number of runs used to construct probability distribution
        complete_threshold: minimum population for a district to be considered complete before appending leftover precincts
        min_complete_count: number of contiguous districts desired
    '''

    # Stores the number of runs in which the Democratic party wins k out of n districts
    # This will be used to generate the probability distribution
    d_wins = {k: 0 for k in range(n + 1)}

    j = 0
    while j < num_runs:
        c_adj = {}
        c_p_adj = {}
        p_adj = {}

        for p in precincts:
            p.district = None

        # Keeps track of unchosen counties/precincts respectively
        c_unchosen = [c.id for c in counties]
        unchosen = [p.id for p in precincts]

        # Copy in contents of the adjacency lists at the beginning of each run
        for c in counties:
            c_adj[c.id] = c_c_adjacency[c.id].copy()
            c_p_adj[c.id] = c_p_adjacency[c.id].copy()
        for p in precincts:
```

```
    p_adj[p.id] = p_adjacency[p.id].copy()

# Initialize the districts
districts = [District() for i in range(n)]
for d in districts:
    p = rand.choice(unchosen)
    node = counties[county_ind[precincts[p].county_name]]

    if node.population > m/n:
        d.phase = 1
        node = precincts[p]

    initialize(counties, d, node, c_unchosen, unchosen, c_adj, c_p_adj, p_adj)

# Continue district construction
complete = True
for d in districts:
    if not d.complete:
        complete = False
while not complete:
    for d in districts:
        if not d.complete:
            add_unit(districts, d, c_unchosen, unchosen, c_adj, c_p_adj, p_adj, m, n)
    complete = True
    for d in districts:
        if not d.complete:
            complete = False

# Append remaining precincts
count = len([d for d in districts if d.population > complete_threshold])
min_pop = min([d.population for d in districts])
if count >= min_complete_count:
    remaining = []
    remaining += unchosen
    new_dist = [d for d in districts if d.population < complete_threshold]

    new_dist_ind = [districts.index(d) for d in districts if d.population < complete_threshold]

    # If all districts are sufficiently populated, pick district with minimum population to add remaining precincts to
    if not new_dist:
        min_dist = districts[0]
        for d in districts:
            if d.population < min_dist.population:
                min_dist = d
        d = min_dist
        while remaining:
            precinct = precincts[remaining[0]]
            d.precincts.append(precinct.id)
            d.population += precinct.population
            d.d_votes += precinct.d_votes
            d.r_votes += precinct.r_votes
            remaining = remaining[1:]

    # If any districts are not sufficiently populated, cycle through these and add remaining precincts
    while remaining:
        for d in new_dist:
            precinct = precincts[remaining[0]]
            d.precincts.append(precinct.id)
            d.population += precinct.population
            d.d_votes += precinct.d_votes
            d.r_votes += precinct.r_votes
            remaining = remaining[1:]

            if not remaining:
                break
    for i in range(len(new_dist)):
        districts[new_dist_ind[i]] = new_dist[i]
```

```
            # A dictionary containing total R and total D votes per district
            count_by_district = get_party_count_by_district(districts)

            # A dictionary {'R': r_votes_in_region, 'D':d_votes_in_region}
            region_count = get_region_count(count_by_district)

            # Add one win to d_wins list at proper index
            d_wins[region_count['D']] += 1

            j += 1

    print_table(d_wins, n, num_runs)

    return districts

# Initializes given district object with chosen precinct or county
def initialize(counties, d, node, c_unchosen, unchosen, c_adj, c_p_adj, p_adj):
    if d.phase == 0:
        for precinct in [p for p in node.precincts]:
            precinct.district = d
            d.precincts.append(precinct.id)
            unchosen.remove(precinct.id)

        c_unchosen.remove(node.id)

        d.c_choices = {x for x in d.c_choices if x != node.id}
        d.c_choices = d.c_choices.union(set(c_adj[node.id])).intersection(set(c_unchosen))

        d.p_choices = {x for x in d.p_choices if x in unchosen}
        d.p_choices = d.p_choices.union(set(c_p_adj[node.id])).intersection(set(unchosen))

        d.population += node.population
        d.d_votes += counties[node.id].d_votes
        d.r_votes += counties[node.id].r_votes

    elif d.phase == 1:
        d.precincts.append(node.id)
        node.district = d
        unchosen.remove(node.id)

        d.current_county = counties[county_ind[node.county_name]]

        d.p_choices = {x for x in d.p_choices if x != node.id}
        d.p_choices = d.p_choices.union(set(p_adj[node.id])).intersection(set(unchosen))

        d.population += node.population
        d.d_votes += precincts[node.id].d_votes
        d.r_votes += precincts[node.id].r_votes

# Used in construction process to add new counties or precincts to districts
def add_unit(districts, d, c_unchosen, unchosen, c_adj, c_p_adj, p_adj, tot_population, num_districts):
    if d.phase == 0:
        # Take intersection with c_unchosen to remove any choices taken by previous districts
        d.c_choices = d.c_choices.intersection(set(c_unchosen))
        if d.c_choices == set():
            d.phase = 1
            return

        node = counties[rand.choice(tuple(d.c_choices))]
        pop = sum([p.population for p in node.precincts if p.id in unchosen])
        if pop + d.population > tot_population/num_districts:
            d.current_county = node
            d.phase = 1

        if d.phase == 0:
            for precinct in [p for p in node.precincts if p.id in unchosen]:
                precinct.district = d
                d.precincts.append(precinct.id)
```

```
                unchosen.remove(precinct.id)

            c_unchosen.remove(node.id)

            d.c_choices.remove(node.id)
            d.c_choices = d.c_choices.union(set(c_adj[node.id])).intersection(set(c_unchosen))

            d.p_choices = {x for x in d.p_choices if x in unchosen}
            d.p_choices = d.p_choices.union(set(c_p_adj[node.id])).intersection(set(unchosen))

            d.population += pop
            d.d_votes += counties[node.id].d_votes
            d.r_votes += counties[node.id].r_votes

    elif d.phase == 1:
        # Take intersection with unchosen to remove any choices taken by previous districts
        d.p_choices = d.p_choices.intersection(set(unchosen))
        if d.p_choices == set():
            d.complete = True
            return

        from_county = [p for p in d.p_choices if precincts[p].county_name == d.current_county.name]
        if from_county:
            node = precincts[rand.choice(tuple(from_county))]
        else:
            node = precincts[rand.choice(tuple(d.p_choices))]

        if precincts[node.id].population + d.population > tot_population/num_districts:
            d.complete = True

        if not d.complete:
            d.precincts.append(node.id)
            node.district = d
            unchosen.remove(node.id)

            d.p_choices = {x for x in d.p_choices if x != node.id}
            d.p_choices = d.p_choices.union(set(p_adj[node.id])).intersection(set(unchosen))

            d.population += node.population

            d.d_votes += precincts[node.id].d_votes
            d.r_votes += precincts[node.id].r_votes

def get_party_count_by_district(districts):
    '''
    Taking in a list of districts, returns a dictionary with keys corresponding
    to districts and values that are lists containing total R votes, D votes
    '''
    return {districts.index(d):[d.r_votes, d.d_votes] for d in districts}

def get_region_count(count_by_district):
    '''
    Taking in R and D count by district, returns a dictionary containing
    total R and D count for the region (all districts)
    '''
    region_count = {'R':0, 'D':0}
    for d in count_by_district.keys():
        if count_by_district[d][0] > count_by_district[d][1]:
            region_count['R'] += 1
        elif count_by_district[d][0] < count_by_district[d][1]:
            region_count['D'] += 1

    return region_count

def print_table(d_wins, n, num_runs):
    '''
    Prints the frequency table
```

```
    Notice that:
        i: number of D wins in a run
        n - i: number of R wins in a run
        d_wins[i]: frequency (number of times democrats had 'i' wins)
        d_wins[i]/num_runs: probability
    '''
    print('D\t\t\tR\t\t\tF\t\t\tP')
    for i in range(len(d_wins)):
        print('{}\t\t\t{}\t\t\t{}\t\t\t{}'.format(i,n-i,d_wins[i],d_wins[i]/num_runs))

def plot_map(districts):
    '''
    Plots districts
    '''
    # plot generation code: https://gis.stackexchange.com/questions/131716/plot-shapefile-with-matplotlib
    plt.figure()

    colors = ["#000000","#4E321D","#C48B5F","#366C28","#87ED6E","#cc1919","#FF0F0F","#FF8FCE", "#80007D", "#FFB3FE",
"#8C8C8C",
              "#8A0000","#FF531A","#1A21FF","#00058F","#8F93FF","#A38000","#FFFF00"]

    print()
    print("Plotting districts...")

    num_precincts = sum([len(d.precincts) for d in districts])
    print(num_precincts)
    for j in range(len(districts)):
        col_idx = j % 18
        d = districts[j]
        choice = colors[col_idx]
        d.color = choice
        for precinct in d.precincts:
            shape = sf.shapeRecords()[precinct]
            x = [i[0] for i in shape.shape.points[:]]
            y = [i[1] for i in shape.shape.points[:]]
            plt.plot(x,y, color=choice, linewidth=0.5)
    plt.show()


def plot_map_party(districts):
    '''
    Plots red-blue party map based on district winners
    '''
    # plot generation code: https://gis.stackexchange.com/questions/131716/plot-shapefile-with-matplotlib
    plt.figure()

    print()
    print("Plotting districts...")

    num_precincts = sum([len(d.precincts) for d in districts])
    print(num_precincts)
    for j in range(len(districts)):
        d = districts[j]
        choice = "#0000FF"
        if d.d_votes < d.r_votes:
            choice = "#FF0000"
        d.color = choice
        for precinct in d.precincts:
            shape = sf.shapeRecords()[precinct]
            x = [i[0] for i in shape.shape.points[:]]
            y = [i[1] for i in shape.shape.points[:]]
            plt.plot(x, y, color=choice, linewidth=0.5)
    plt.show()

if __name__ == "__main__":
    '''
    Creating distribution for 2016 congressional election with
        m: population of Pennsylvania (sum of population of precincts),
```

```
    n: 18 districts,
    num_runs: 100,
    complete_threshold: 670000 (roughly 80% of m/n)
    min_complete_count: 15 districts
'''
dist = main(m = 12732284, n = 18, num_runs = 100, complete_threshold=670000, min_complete_count=15)
plot_map(dist)
plot_map_party(dist)
```

BIBLIOGRAPHY

2016 Pennsylvania House Election Results. Politico. (n.d.). Retrieved March 30, 2022, from https://www.politico.com/2016-election/results/map/house/pennsylvania/

Adams, B. & Netznik, N. (2021). Monte Carlo Simulation to Estimate Probability Distributions of Party Representation in Political Redistricting. Proceedings of the Northeast Decision Sciences Institute

Alexeev, B., & Mixon, D.G. (2018). Partisan gerrymandering with geographically compact districts. *Journal of Applied Probability, 55,* 1046-1059. doi：10.1017/jpr.2018.70

Barkstrom, J., Dalvi, R., & Wolfram, C. (2018). Detecting gerrymandering with probability: A markov chain monte carlo model. (Unpublished paper）

Chambers, C. P. & Miller, A. D. (2010). A Measure of Bizarreness. *Quarterly Journal of Political Science, 5(1),* 27-44. doi:http://dx.doi.org/10.1561/100.00009022

Chatterjee, T., DasGupta, B., Palmieri, L., Al-Qurashi, Z., & Anastasios, S. (2020). On theoretical and empirical algorithmic analysis of the efficiency gap measure in partisan gerrymandering. *Journal of Combinatorial Optimization, 40*, 512-546. doi:https://doi.org/10.1007/s10878-020-00589-x

Chen, J. (2017). The Impact of Political Geography on Wisconsin Redistricting: An Analysis of Wisconsin's Act 43 Assembly Districting Plan. *Election Law Journal, 16*(4), 443-452. doi:10.1089/elj.2017.0455

Chen, J. & Cottrell, D. (2016). Evaluating partisan gains from Congressional gerrymandering: Using computer simulations to estimate the effect of gerrymandering in the U.S. House. *Electoral Studies, 44,* 329-340. doi:https://doi.org/10.1016/j.electstud.2016.06.014

Chen, J., & Rodden, J. (2013). Unintentional Gerrymandering: Political Geography and Electoral Bias in Legislatures. *Quarterly Journal of Political Science, 8,* 239-269. doi:10.1561/100.00012033

Cirincione, C., Darling, T. A., O'Rourke, T. G. (2000). Assessing South Carolina's 1990s congressional redistricting. *Political Geography, 19*, 189-211. doi:https://doi.org/10.1016/S0962-6298(99)00047-5

Common Cause et al. v. Lewis et al. (2019). https://www.nccourts.gov/assets/inline-files/18-CVS-14001_Final-Judgment.pdf?Bwsegeo1VV20zhJsp9hoClvmoRp3A6AR

Cooper, W., Seiford, L., & Zhu, Joe. (2011). Data Envelopment Analysis: History, Models, and Interpretations. doi:10.1007/978-1-4419-6151-8_1

Cotz, G. W. & Katz, J. N. *Elbridge Gerry's salamander: The electoral consequences of the reapportionment revolution.* Cambridge University Press.

Criss, D. (2019, June 27). *Gerrymandering – explained*. CNN. https://www.cnn.com/2019/06/27/politics/what-is-gerrymandering-trnd

Davis et al. v. Bandemer et al., 478 U.S. 109 (1986). https://tile.loc.gov/storage-services/service/ll/usrep/usrep478/usrep478109/usrep478109.pdf

Fifield, B., Higgins, M., Imai, K. & Tarr, A. (2020). Automated Redistricting Simulation Using Markov Chain Monte Carlo. *Journal of Computational and Graphical Statistic*s, *29*(4), 715-728. doi:10.1080/10618600.2020.1739532

Fryer, R. G., Jr., & Holden, R. (2011). Measuring the Compactness of Political Districting Plans. *The Journal of Law & Economics, 54*(3), 493-535. doi:10.1086/661511

Herschlag, G., Kang, H. S., Luo, J., Graves, C. V., Bangia, S., Ravier, R., & Mattingly, J. C. (2020). Quantifying Gerrymandering in North Carolina. *Statistics and Public Policy, 7*(1), 30-38. doi:https://doi.org/10.1080/2330443X.2020.1796400

Hodge, J. K., Marshall, E., & Patterson, G. (2010). Gerrymandering and Convexity. *College Mathematics Journal*, *41*(4), 312-324. doi:10.4169/074683410X510317

Kang, M. S. (2020). Hyperpartisan Gerrymandering. *Boston College Law Review*, *61*(4), 1379-1445. https://lawdigitalcommons.bc.edu/bclr/vol61/iss4/4

Kaufman, A., King, G. & Komisarchik, M. (2020). How to measure legislative district compactness if you only know it when you see it. (Unpublished paper).

League of United Latin American Citizens v. Perry, 548 U. S. ____ (2006).

League of Women Voters of Pennsylvania et al. v. the Commonwealth of Pennsylvania et al. (2018). http://www.pacourts.us/assets/files/setting-6061/file-6852.pdf?cb=df65be

Maceachren, A. (1985). Compactness of Geographic Shape: Comparison and Evaluation of Measures. Geografiska Annaler. Series B, Human Geography, 67(1), 53-67. doi:10.2307/490799

Merriam-Webster (n.d.). Gerrymandering. In Merriam-Webster.com dictionary. Retrieved March 24, 2021, from https://www.merriam-webster.com/dictionary/gerrymandering

Miller et al. v. Johnson et al., 515 U.S. 900 (1995). https://tile.loc.gov/storage-services/service/ll/usrep/usrep515/usrep515900/usrep515900.pdf

PA Const. art. II, § 16

PA Const. art. II, § 17

Pennsylvania House of Representatives elections, 2016. Ballotpedia. (n.d.). Retrieved March 30, 2022, from https://ballotpedia.org/Pennsylvania_House_of_Representatives_elections,_2016

Reynolds et al. Sims et al., 377 U.S. 533 (1964).

Rucho et al. v. Common Cause et al., 588 U.S. ____ (2019). https://www.supremecourt.gov/opinions/18pdf/18-422_9ol1.pdf

Shaw et al. v. Reno et al., 509 U.S. 630 (1993).

Stephanopoulos, N., & McGhee, E. (2015). Partisan Gerrymandering and the Efficiency Gap. *The University of Chicago Law Review, 82*(2), 831-900. Retrieved March 9, 2021, from http://www.jstor.org/stable/43410706

Tapp, K. (2019) Measuring Political Gerrymandering. *The American Mathematical Monthly, 126*(7), 593-609, doi: 10.1080/00029890.2019.1609324

U.S. Const. art. I, § 2

Vieth et al. v. Jubelirer et al., 541 U. S. ____ (2004). https://www.supremecourt.gov/opinions/03pdf/02-1580.pdf

Voting Rights Act of 1965, 52 U.S.C. §10101 et seq. (1965).

Wang, S. S. H. (2016). Three tests for practical evaluation of partisan gerrymandering. *Stanford Law Review, 68*(6), 1263-1289. https://link.gale.com/apps/doc/A460507850/LT?u=carl39591&sid=LT&xid=ad257f5e

Whitford et al. v. Gill et al. (2016). http://www.lb7.uscourts.gov/documents/16-1161-op-bel-dist-ct-wisc.pdf

ACADEMIC VITA

Education:

Bachelor of Science Degree in Mathematical Sciences, Penn State Harrisburg, Spring 2022

Bachelor of Science Degree in Computer Science, Penn State Harrisburg, Spring 2022

Honors in Mathematical Sciences

Thesis Title: Evaluating Redistricting in Pennsylvania using Monte Carlo Simulation

Thesis Supervisor: Dr. J. Brian Adams

Faculty Reader: Dr. Jeremy Blum

Honors Adviser: Dr. Ronald Walker

Experience:

Peer Tutoring, Russell E. Horn Sr. Learning Center, Fall 2019 – Spring 2022

Supervisor: Lainey Schock

Ohio State University MBI REU Program, Summer 2020

Supervisors: Dr. Enkeleida Lushi, Dr. Kristen Severi

Honors Service Learning at WE cARE Food Pantry, Fall 2021

Supervisor: Ashley Schools

Awards:

Dean's List, Fall 2017 – Spring 2022

President's Freshman Award, Spring 2018

Mathematical Science Outstanding Undergraduate Student Award, Spring 2022

Activities/Presentations:

Pi Mu Epsilon (Vice President), Upsilon Pi Epsilon, Phi Kappa Phi

Presenter, Paper titled "Monte Carlo Simulation to Estimate Probability Distributions of Party Representation in Political Redistricting", Northeast Decision Sciences Institute 2021 conference

Co-author, Paper titled "Estimating the Probability Distribution of Party Representation as a Result of Political Redistricting Using a Random Walk Monte Carlo Technique", Journal of Management Policy and Practice, Summer 2021