Statistics, Politics, Law and Gerrymandering

Amar Khullar
201487417

Supervised by Robert Aykroyd

Submitted in accordance with the requirements for the
module MATH5871M: Dissertation in Statistics
as part of the degree of

Master of Science in Data Science and Analytics

The University of Leeds, School of Mathematics

September 2021

The candidate confirms that the work submitted is his/her own and that appropriate credit has been given where reference has been made to the work of others.

Abstract

The purpose of this dissertation is to investigate gerrymandering in the United States, specifically in Massachusetts. This involves researching the methods used by gerrymanderers as well as the difficulties concerned in reliably detecting gerrymandering fairly. I created a data-set through manual encoding combined with merging existing data sets, then used Markov Chain Monte Carlo, a statistical sampling method to make an assessment on whether Massachusetts has been gerrymandered. Furthermore, I evaluate the different proposal functions used in Markov Chain Monte Carlo for redistricting: boundary flip and recombination of districts, and compare their results on my data-set.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background and history

Every decade in the United States, each state redraws its electoral district boundaries to account for changes in population size and density. In most states the state legislature draws the lines that split the state into districts, and in recent news it has been proven that a few states have redistricted their state unfairly, with partisan intent. This is called gerrymandering, and mathematicians have been testifying in court to stop these biased maps from being used.

The term gerrymandering arose in 1812, when governor of Massachusetts Elbridge Gerry signed off on a new district map, drawn by the Democratic-Republican legislature. The map that helped gain the Democratic-Republican party an unproportionable number of seats was so distorted that it resembled a salamander, hence the local newspapers coined the term "Gerrymander".

Figure 1.1: Massachusetts' congressional district resembling a salamander.

Unfortunately gerrymandering is still prevalent in modern times and has been made easier through the use of computer slicing. The way gerrymanderers create unfair maps is by 'packing and cracking'. They 'pack' many of their opponents votes into a few districts, and 'crack' the rest of their opponents votes sparsely over many districts (Lapowski 2018). This allows their opponents to overwhelmingly win a few seats, but lose the majority number of seats, hence losing the state. Figure 2 shows the effect that redistricting can have on the outcome of an election.

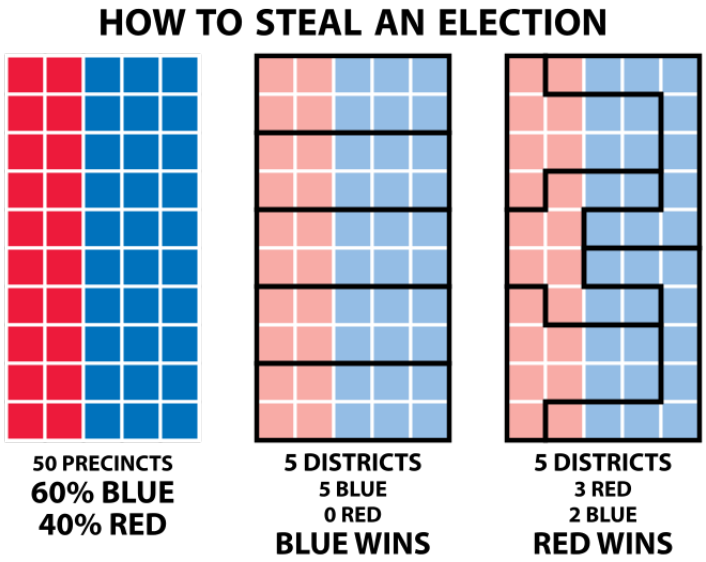


Figure 1.2: How to steal an election

The blue party has 60% of the votes, however the number of seats each party wins differs drastically depending on the district boundaries drawn. The second map even allows the red party to win, despite the strong blue majority in the popular vote. The map is not proportionally representing the vote share.

Proportional representation is when the number of seats won by each party is proportional to the number of votes the party won (Ellenberg 2020). It is one of the original criteria used to determine the fairness of a district map.

Ideally, all district maps would proportionally represent their voters, however this is impossible due to geography, uniform voting patterns and the nature of the world. The best we can do is eliminate the obviously biased maps. Even apparently fair maps do not always achieve proportional representation. Consider the map in the middle of figure 1.2, the blue party wins all five seats, however if the map was representing each party proportionally, the blue party should have won 3 seats.

1.2 Understanding the problem

The problem that makes gerrymandering hard to detect is that disproportional outcomes do not automatically mean that there has been gerrymandering. If a candidate received 35% of the votes but no seats, this does not necessarily indicate that there has been gerrymandering. If voters are not clustered, seats will not be won. Any voting minority needs a certain amount of non-uniformity in how it's votes are dispersed to gain seats.

Another dilemma is that detecting gerrymandering can be complicated; if a map is contorted this could be due to several reasons, such as following county lines or preserving communities. Failing in the past, judges have deemed these maps' fairness by eye. Nearly 37 states still use the 'eyeball test' to judge the fairness of these maps (Duchin 2018a). A new standard for assessing the fairness of district maps is essential.

Recently, classical metrics have failed us too. In 2018 the League of Women Voters took the Pennsylvanian legislature to court accusing them of gerrymandering their congressional district map. The map scored well on the existing rules set in place: Districts were contiguous, equally distributed and achieved strong compactness scores in five specified formulae (Duchin 2018a). However, upon further inspection and use of statistics, mathematicians proved to the judge that the map contained severe partisan bias. They did this through Markov Chain Monte Carlo, a statistical sampling method that uses a random walk to assess the partisan bias of the map compared to potential alternatives.

This works by taking the current congressional district map and slightly modifying it to create a marginally different map, however one that still satisfies the government criteria. In most states these criteria are: Compactness, Contiguity and equal population. The new generated maps are then compared to the original map using metrics. Some of the metrics include Efficiency gap, Reock score, Popper-Polsby measure and partisan symmetry. These metrics work based on the number of wasted votes, compactness of the districts, and voting history.

In an ideal world, we would have all the possible district maps and compare all metrics in each of these possible maps to judge a new map's fairness, however the space of possible district maps is so large that this would be an intractable problem. Markov Chain Monte Carlo utilizes Ergodic theorem, which states that if you random walk for a significant amount of time, the collection of generated maps will contain the properties representative of the entire space of possible maps (Duchin 2018a). This remains true even when you have only random walked through a fraction of the whole space of maps. This theorem allows us to determine if the map you have is an extreme outlier according to our political metrics.

1.3 Overview of the project

I have decided to bring scrutiny back to Massachusetts, where gerrymandering first began. Recently it has mainly been Republican states in the news being accused of gerrymandering, so an analysis of a highly democratic state could shed some new information. Democrats have won every single electoral seat in Massachusetts since 1994. The goal of this dissertation is to try and determine whether they have been unfairly redrawing their maps in attempt to maintain their hold.

This task involves collecting, annotating, and collating a handful of data-sets and using Markov Chain Monte Carlo to assess whether the current map has partisan bias. The partisan bias will be determined through calculation of several metrics, both on the original data and on modified data. 65% of the votes in the last election were for the Democrats, so I may need to modify the data to be more balanced, and test on the modified data-sets to gain more useful insights. I will also analyze two different proposal functions used in Markov Chain Monte Carlo for redistricting, to try and gain some insight into their useful properties and use cases.

1.4 Useful terminology

The following terms will be used throughout the report:

- Precinct – the smallest possible unit of voting land
- Municipality – Most municipalities contain more than one precinct, however some only contain one. An example of a municipality would be Abington, which contains five precincts
- District – A group of contiguous precincts. In each district, whichever party has the most votes wins a seat
- Wasted votes – If a party wins the district, then it's wasted votes are the difference between the votes needed and the votes received. If a party loses a district then it's wasted votes are the votes they received.
- MCMC – Markov Chain Monte Carlo
- Dual graph – a mathematical graph representation of the state of Massachusetts. Each node corresponds to a unique municipality, and an edge represents that the municipalities are adjacent
- Vertex/Node – An element of a graph which connects to other vertices/nodes through edges
- Edge – A link between vertices/nodes

Chapter 2

Theory: Markov Chain Monte Carlo for Redistricting

2.1 Markov Chain Monte Carlo

Markov Chain Monte Carlo (MCMC) is a method of sampling from an unknown distribution. It has been used by statisticians for decades to simulate posterior distributions. The benefit of using MCMC is that you can accurately estimate useful properties of posterior distributions which would otherwise be intractable to calculate (Kass 1998).

2.1.1 Markov Chains

It utilises the idea of Markov chains, which are stochastic processes where the next state in the process is dependent entirely on the current state. A way of visualizing this is shown in figure 2.1, as a mathematical graph.

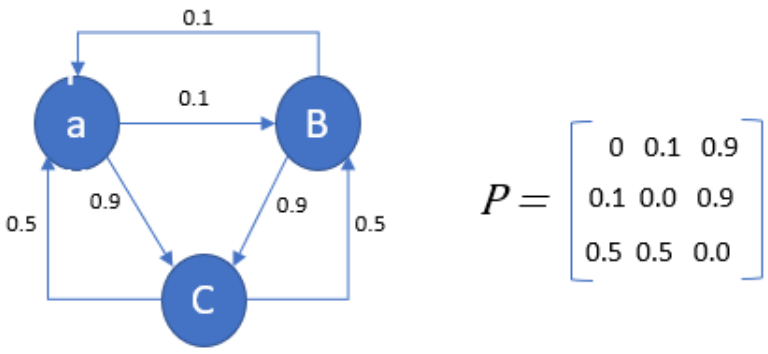


Figure 2.1: Example Markov Chain

The graph is represented by nodes A,B,C, and weighted edges connecting them. P is the state transition matrix / Markov distribution.

If you are in state C, then you have a 50% chance of transitioning into state A, and a 50% chance of transitioning into state B. The transition function is memoryless, it is not concerned with how the current state was reached. Another way to describe this process is a 'Random walk'.

In practice, we might not know the transition probabilities for all our states, but transition probabilities can be recorded to create Markov distributions. A popular use for this type of process is in text autocomplete, for example if the current state is 'worl', there would be a high transition probability to the state 'world'.

Definition - Irreducibility: A Markov chain is irreducible if any state can be reached from any other state, in a finite number of steps.

Definition - Periodicity: A state is periodic if the Markov chain can only return to said state by taking more than 1 transition. A Markov chain is aperiodic if every state is not periodic (DeFord 2019).

Periodicity can be implemented in a Markov chain by simply adding a probability of standing still to each state.

A Markov chain is ergodic if it is irreducible and aperiodic. Ergodicity provides a very important property, which states that there is a unique stationary distribution, and regardless of the initial state of the Markov chain, this stationary distribution will be reached with enough iterations of the transition function. In many applications, an objective function is used to identify the stationary distribution, however this is not feasible in the case of redistricting (DeFord 2020).

The number of iterations to reach the stationary distribution is called the Markov chain's mixing time. Experimentally, convergence metrics are normally used to decide when you have reached close enough to the stationary distribution.

2.1.2 Monte Carlo sampling

Monte Carlo sampling is a method of sampling from a probability distribution to calculate some derived values. Consider rolling two dice to estimate the probability of rolling a sum of eight. You could calculate this manually by summing the probabilities of all the possible ways to get a total of eight, or by rolling two dice a thousand times and seeing how often an eight occurs. This example is trivial, but the same method can be applied when calculating the probability mathematically is intractable.

We combine these ideas together to get MCMC, which we can use to attempt to solve the redistricting problem. The goal is to create an irreducible, aperiodic Markov chain, to sample from a desirable stationary distribution in the space of possible district maps. With these samples we can calculate valuable metrics to compare with the original map, with the purpose of assessing partisan bias.

The benefit of using a Markov chain in sampling is that we can sample from a distribution

without knowledge of the probabilities connected to it (DeFord 2019). This is done by sampling proportionally to a score function using the Metropolis-Hastings algorithm.

Metropolis-Hastings MCMC requires a proposal function, a score function and an acceptance function, and it works through acceptance-rejection sampling. At each iteration, you generate a new sample through the proposal function and decide whether or not to accept it into the Markov chain depending on it's 'score'. Since a score threshold is required, this makes our Markov chain aperiodic, because there is a chance of remaining in the current state.

2.2 Markov Chain Monte Carlo as a redistricting problem

MCMC is recognized as a great tool for approximating distributions, but only recently has it been used for redistricting. There are challenges in development of an algorithm that samples from a desirable stationary distribution, and there is contention about the best method of doing this. The choice of proposal and acceptance functions is very important. In practice we cannot see or calculate our stationary distribution, so researchers have drawn conclusions through experimental testing. Moreover, we must consider what derived values are useful to calculate in our samples to assess partisan bias.

Combining the redistricting problem with MCMC will require the following steps:

1. Creating a dual graph for Massachusetts
2. Define what a valid plan is using Massachusetts state laws
3. Proper choice of proposal function to sample through possible maps
4. Create acceptance function for desirable maps

2.2.1 Dual graph

A dual graph is an undirected, unweighted mathematical graph which uses vertices and edges to encode the information we want.
We can define our graph as:

$G = (V, E)$

Where $V = \{v_1, v_2, \ldots, v_n\}$ is a set of vertices

$E \subseteq \{\{u,v\}$ where $u, v \in V$ and $u \neq v\}$ is a set of vertices pairs

In the case of redistricting, vertices contain the information about the geographic block it represents.
The graph must be fully connected to satisfy the irreducibility property required for ergodicity.

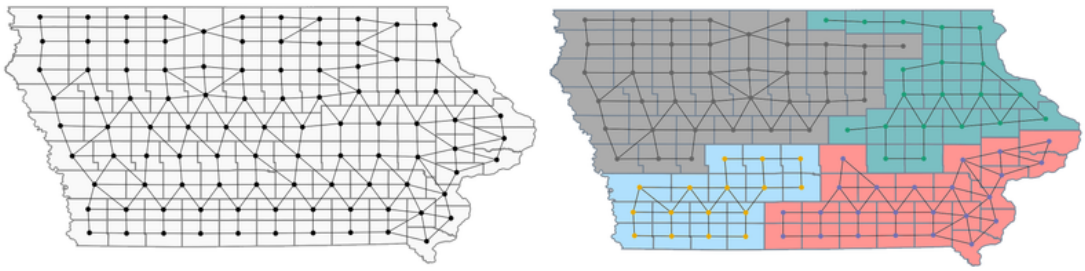


Figure 2.2: Dual graph of Iowa

Figure 2.2 shows a dual graph representation for the state of Iowa. There are multiple ways of creating this graph, you could have a node represent a whole county, a municipality or a single precinct. This choice will affect both the time it takes to run MCMC due to the size of the graph, and the quality of results.

By setting up the problem with a dual graph, it also allows us to perceive the problem of assigning districts as a graph partitioning problem (Fifield 2020).

2.2.2 Validity constraints

The state of Massachusetts requires their districts to be equally populated, of contiguous territory, without dividing towns or wards (Legislature 2021).

Massachusetts requires districts to have the exact same population to the +- 1 person. This is not feasible to implement in MCMC so I will be using a threshold for population balance. Consider if there are 100 people in a district with a 5% population threshold, a change can only be made that changes the population to between 95-105, any change outside of this range would be rejected.

Dividing towns or wards is a problem that will be considered when deciding how in depth the dual graph is. If a precinct level is used, this will need to be programmed into the acceptance function. Otherwise, at municipality and county level, this condition will automatically be satisfied.

Contiguity can also be encoded into the acceptance function. This condition will be affected uniquely by different proposal functions.

2.2.3 Proposal functions

There are two popular proposal functions for redistricting. The boundary flip proposal, and a more recent innovation, the recombination of districts proposal. They work in an inherently different manner to each other, and there are differences in the time at which they converge to the stationary distribution, as well as the properties represented in the samples (Duchin 2018a).

2.2.4 Boundary Flip

The older method of implementing MCMC for redistricting is the boundary flip method. This works through iteratively changing the district of a single node on a district boundary, while preserving the validity rules (DeFord 2019). There are several subtly different ways you could implement this, however each will produce a different stationary distribution. One method follows the process outlined:

1. Randomly sample a node that is on a district boundary (This can be done uniformly across districts, or proportionally to the number of nodes in each district)
2. With a 50% chance, change the district of the sampled node to match the district of its neighbour in the other district. With a 50% chance change the district of the neighbour to match the district of the sampled node

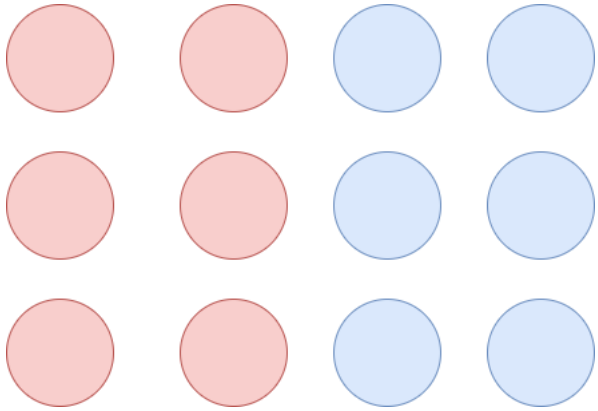


Figure 2.3: Example graph before boundary flip

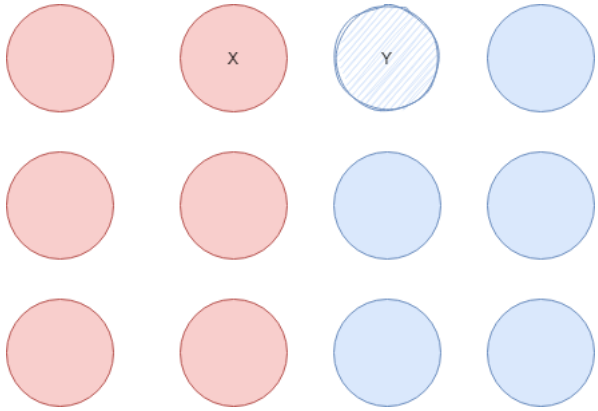


Figure 2.4: Sampling a boundary node

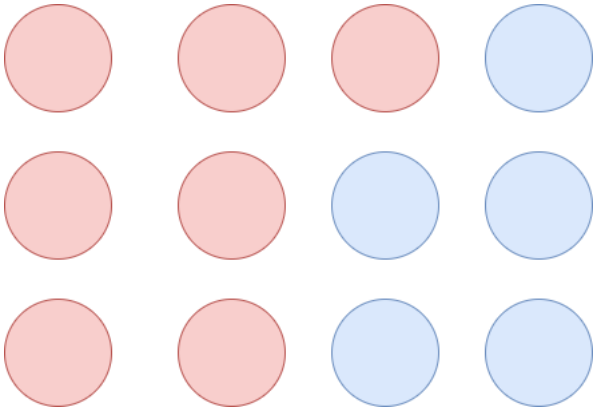


Figure 2.5: Example graph after successful boundary node iteration

Figures 2.3 - 2.5 show a simple example of the boundary flip proposal. Node Y, highlighted, is the boundary node that has been sampled. Node X and node Y are candidates for a district change with a 50% chance. In this case, node Y has been selected and has changed district.

This proposal is intuitive to understand and to demonstrate it's Markov property; being that the next state depends entirely on the current state. It is also easy to explain to non-technical people, which is important in legal trials. A problem that may arise with this proposal is contiguity, as there is no guarantee that a flip will not break apart a district into two separate components.

Regarding the experimental use of the algorithm, it is relatively easy to implement, but it is reported to have slow mixing times. Duchin noted that 'this Flip ensemble is far from convergence after a billion steps' on a 100x100 grid (Duchin 2018a).

Furthermore, experiments lead us to believe that boundary flip produces 'long-winding' stationary distributions. This means that sampled maps tend to have long, thin, and chaotic districts, rather than compact districts.

The reason why this proposal creates so many chaotic districts is because of the sheer number of them. Considering the space of all maps, the number of districts that conform to our idea of a 'normal' map is a tiny subset, and the space is dominated by random-looking configurations. Nonetheless, a well-constructed acceptance function which prefers compact maps can help produce meaningful results with this proposal.

2.2.5 Recombination of districts

A novel method in the space of redistricting proposals is called Recombination of districts. It was published in 2020 by some of the key contributors to the mathematics of gerrymandering, Duchin, DeFord and Solomon. This proposal examines redistricting as a graph partitioning problem, where the random walk is through the space of graph partitions. The method also attempts to fix the issue with chaotic maps, by only sampling maps that are desirably shaped.

Recombination works through the following steps:

1. Sample two districts that neighbour each other (Sample proportionally to the length of the district boundary to improve compactness)
2. Form an induced sub-graph for those two districts
3. Generate a random spanning tree for the induced sub-graph
4. Iteratively 'cut' an edge from the spanning tree
5. If the two components resulting from the cut satisfy our acceptance function, update the graph to reflect the new districts from the cut

To help understand every step of this process I will note some important definitions in graph theory:

Definition - Cycle: A cycle in a mathematical graph is a subset of the edge set of the graph such that all edges are distinct, and the first node of the path corresponds to the last (Weisstein 2020).

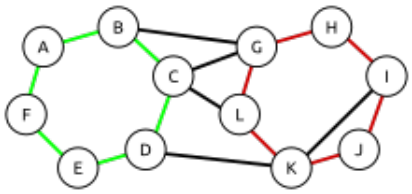


Figure 2.6: Example graph with cycles

Figure 2.6 illustrates a graph with 2 cycles, highlighted by green and red.

Definition - Spanning tree: A spanning tree of a graph is a connected subgraph containing all n vertices but only n-1 of the edges, such that there are no cycles in the subgraph.

A property of spanning trees to note is that one graph can have multiple spanning trees, which all have the same number of edges and vertices. A useful property for MCMC is that if you remove an edge from a spanning tree, the resulting graph will become disconnected.

The figures below demonstrate one iteration of the recombination process, which utilises random spanning trees. A random spanning tree is a spanning tree which has been sampled uniformly from the space of all possible spanning trees.

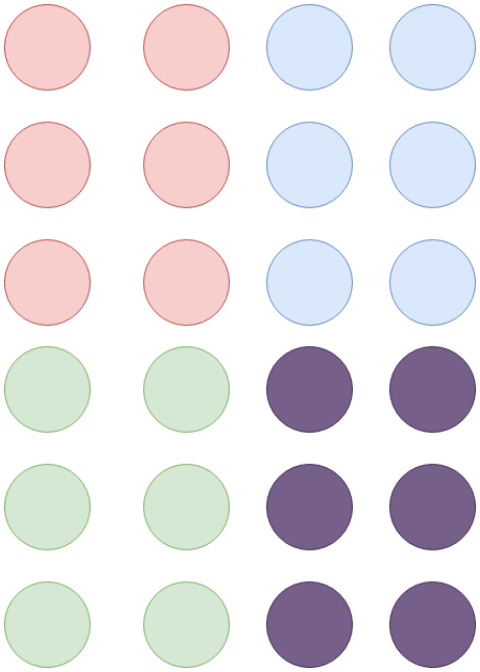


Figure 2.7: Example graph before Recombination iteration

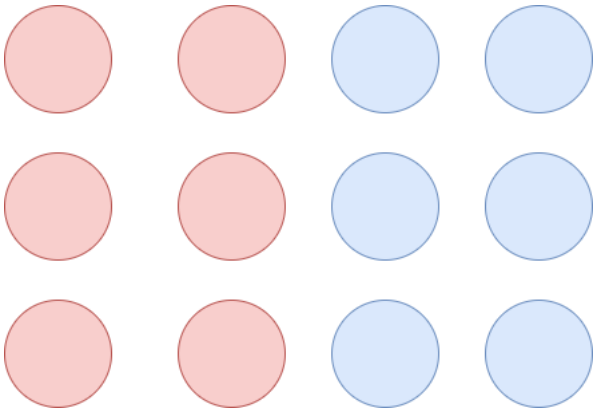


Figure 2.8: Step 1: Sampling two neighbouring districts

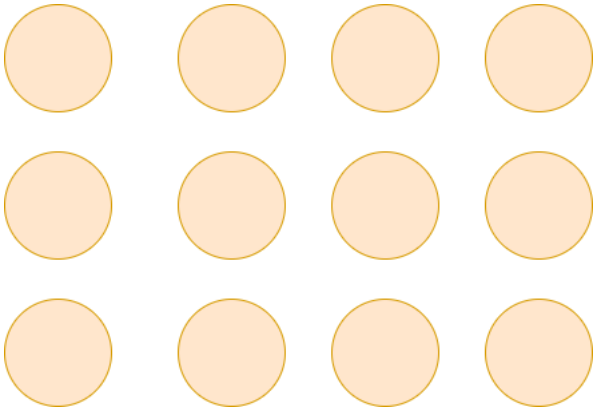


Figure 2.9: Step 2: Form induced sub-graph

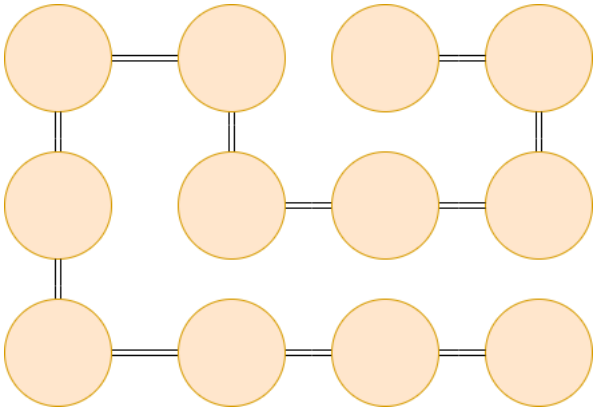


Figure 2.10: Step 3: Generate random spanning tree for sub-graph

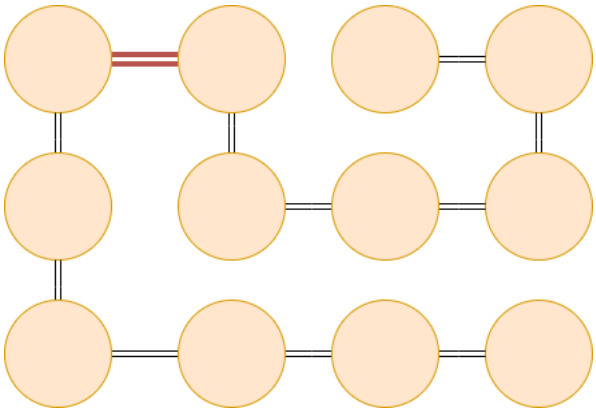


Figure 2.11: Step 4: Cut an edge from the spanning tree

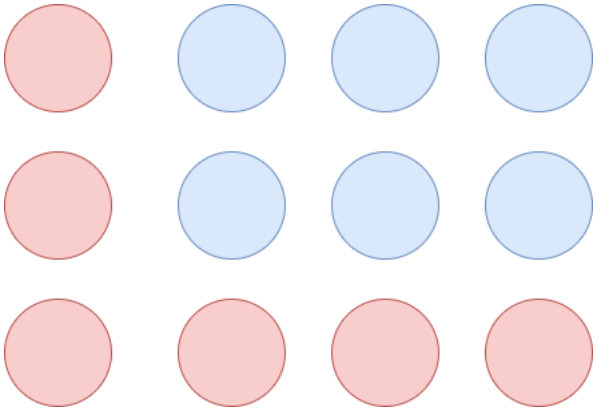


Figure 2.12: Step 5: Update graph to reflect new districts after valid cut

Step 5 will only occur if the cut that was taken in step 4 satisfies the conditions of our acceptance function.

One can notice how it is much quicker to obtain a wider range of the types of graphs produced through recombination steps, over boundary flip steps. At each iteration we are sampling from the full space of plans, rather than just the space of 'one step away' plans. This quality is even more pronounced with larger graphs.

This is one of the benefits of the recombination method. Each acceptance of a cut from a recombination step brings you much closer to the stationary distribution than a single accepted step from a boundary flip iteration. However, this comes with a limitation, each successful recombination step takes much longer than a boundary flip step so you cannot expect to produce as many different maps in the same amount of time with both methods.

Nevertheless, the quality of maps produced with recombination is far greater and more varied. The method of cutting a spanning tree means that contiguity is automatically implied. Furthermore, by sampling the two neighbouring districts proportionally to boundary size, compact maps can be produced. These two criteria result in more 'normal-looking' maps compared to the boundary flip proposal.

2.2.6 Acceptance function

The score function that is used to determine whether a new proposed plan is accepted will be a function on compactness and population.

Number of boundary nodes, and number of cut edges can all provide a metric for compactness in the graphs. Cut edges are the edges in a graph such that one end of the edge is in a different district to the other end.

There are multiple ways of implementing population score. One method of calculating population score is by comparing the populations of the new two districts in the proposed cut, against the original average population of all districts, where both of the two proposed new districts need to satisfy the criteria. A different way would be to compare the populations of the two sampled districts and attempt to maintain a balance which uses a threshold of the population

difference between the two districts, rather than the average population of the original districts. This alternative method would result in a different stationary distribution, perhaps one with a wider range of population spread.

2.2.7 Metrics

There is contention about the best metrics to record in order to prove gerrymandering. Compactness scores such as Reock Score, the Polsby-Popper measure and the convex hull have been used in court. The Reock score works by comparing the area of a district to a circle, and measuring the difference. It is essentially a measure of circularity. The convex hull works through comparing the district to it's minimum bounding polygon, and the Polsby-Popper measure uses the area/perimeter ratio. The problem with these measures is that they aren't compatible, if a district scores well in one it will score poorly in the others. Consider a round shaped district, it's Reock score would be close to 1, however it's convex hull score would be substandard. Graphical measures such as edge-cuts are more pertinent to dual graph representations, and have been shown to improve compactness efficiently (Dube 2016).

Political metrics are essential for producing proof of partisan bias. Popular metrics in the past have been partisan symmetry and the efficiency gap. Partisan symmetry utilizes a seats-votes curve. The seats-votes curve is simple, the proportion of votes won by the party is on the x-axis, and the proportion of seats won by the party is on the y-axis (Duchin 2018b). One election represents a single point on the curve. The seats-votes curve assumes a two party system. Partisan symmetry is satisfied when the seats/votes curve is invariant under the symmetry:

$(x, y) \to (1 - x, 1 - y)$

This means that if in one year party x gets 60% vote share and wins 4 seats, then in another year if party y gets 60% vote share it should also win 4 seats. Under this restraint, if a party gets 50% vote share they should get exactly half the seats (Ellenberg 2020).

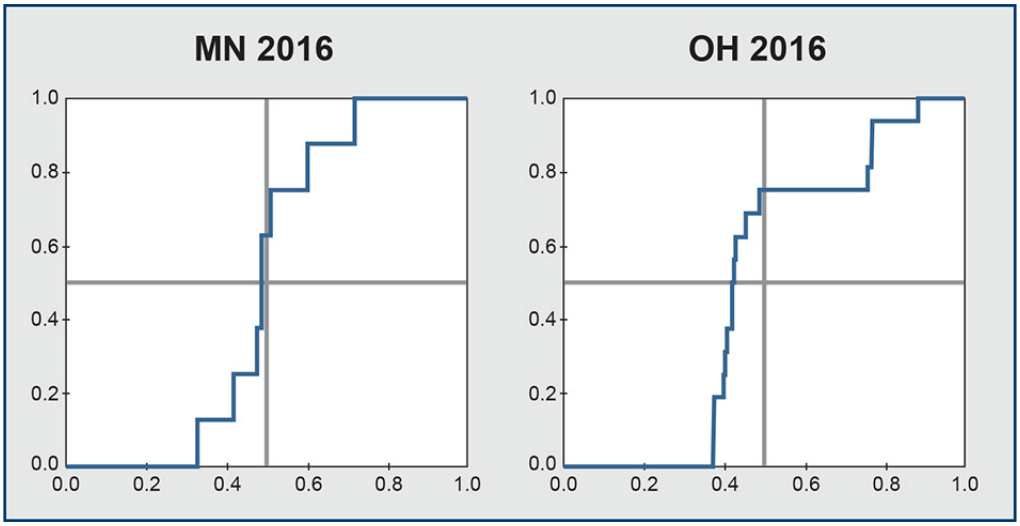


Figure 2.13: Seats-votes curves for Minnesota and Ohio 2016

Figure 2.13 shows the seats-votes curves for Minnesota and Ohio in 2016. Minnesota has a very symmetrical curve, and Ohio has quite an asymmetrical curve, which indicates partisan bias in the district map.

Often in practice, in order to see if a map satisfies partisan symmetry we need to interpolate new data. We do this by adjusting to see what would happen if the vote proportions were different, lets say 50%. We adjust the votes of one party by the same amount in each district then calculate the number of seats that it wins.

However, partisan bias isn't concrete proof of gerrymandering. Partisan asymmetry can occur naturally, due to different areas having different types of voters (Ellenberg 2020). A more modern metric used in gerrymandering is the efficiency gap. The efficiency gap (E) is defined as follows:

$E = (W_a - W_b)/Votes_{Total}$

Where W_a and W_b) are the wasted votes for party A and party B respectively.

Wasted votes, W_x as calculated as follows:

$W_x = \sum_{d=1}^{N_d} W_{x,d}$

Where N_d is the number of districts in the state and

$W_{x,d} = W_{x,d} - (W_{x,d} + W_{y,d})/2$ | if x wins the district

$W_{x,d} = W_{x,d}$ | if y wins the district

Essentially, the efficiency gap is the number of wasted votes for party A minus the number of wasted votes for party B divided by the total number of votes. Wasted votes are the excess votes a party won over the votes required to win (if they won the district), or the total number of votes won (if they lost the district). Efficiency gap is generally a good measure of how fair a district is, (Ellenberg 2020) and as well as this, it doesn't require extrapolation of data, it is calculated with real election data. However, a drawback of the efficiency gap metric is that it can drastically change, with even a small change in vote share. We can workaround this by forming a distribution of efficiency gaps on our ensemble of maps when we perform MCMC, but the results may still reflect this erratic property.

Chapter 3

Methodology

3.1 Creating the data-set

A significant amount of time was involved in creating the data-set, because obtaining data for Massachusetts was an arduous task. There was a surprising lack of existing data-sets that combined congressional district data with precinct and location data. Resultantly, I decided to create a data-set that captures this information, so that others could also use it and provide more perspectives on the topic.

This process required several planned steps. Before carrying them out I tested them out with a small portion of the data to ensure that the data-set would work with MCMC. The steps are as follows:

1. Label each municipality with a global ID on the map
2. Create a file which maps municipality name to ID and district number (For municipalities that are split, separate each into multiple IDs)
3. Read in all precinct names and populations from online Massachusetts statistics PDF
4. Read and sum voting data for all municipalities except the split ones
5. Do the voting data addition manually for the split municipalities
6. Merge together the municipality names / IDs to populations
7. Merge together output of previous step with voting data
8. Label municipality adjacencies in data-set
9. Verify data is correct

To begin I needed to find the most recent map of Massachusetts' congressional districts. Figure 3.1 shows the 2011-2021 congressional district map of Massachusetts.

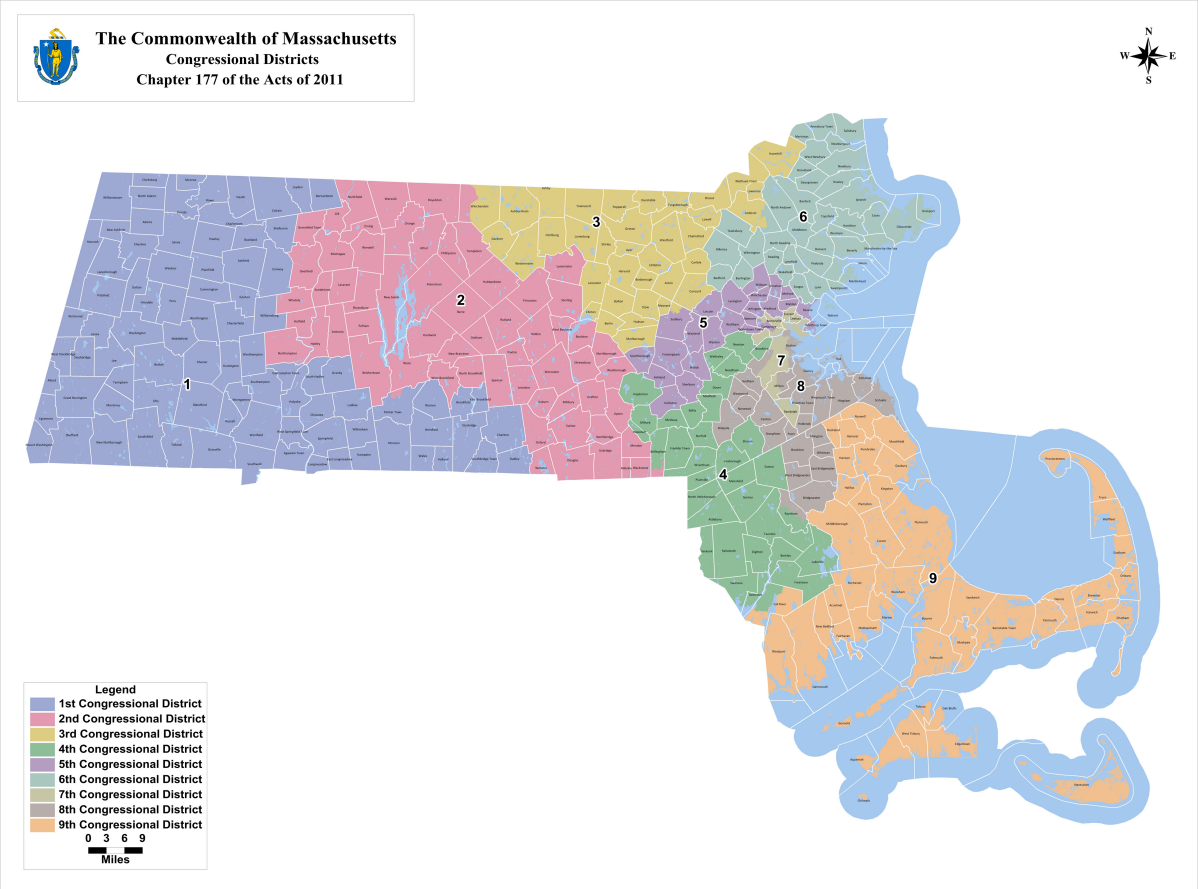


Figure 3.1: Massachusetts congressional district map

I hand labelled the congressional district map and entered the municipality identifiers and names into a data-set. The labelled map is displayed in figure 3.2.

Each municipality is uniquely identified with a number. It is worth noting that many of the municipality's labelled on the map contain more than one precinct, however some only contain one.

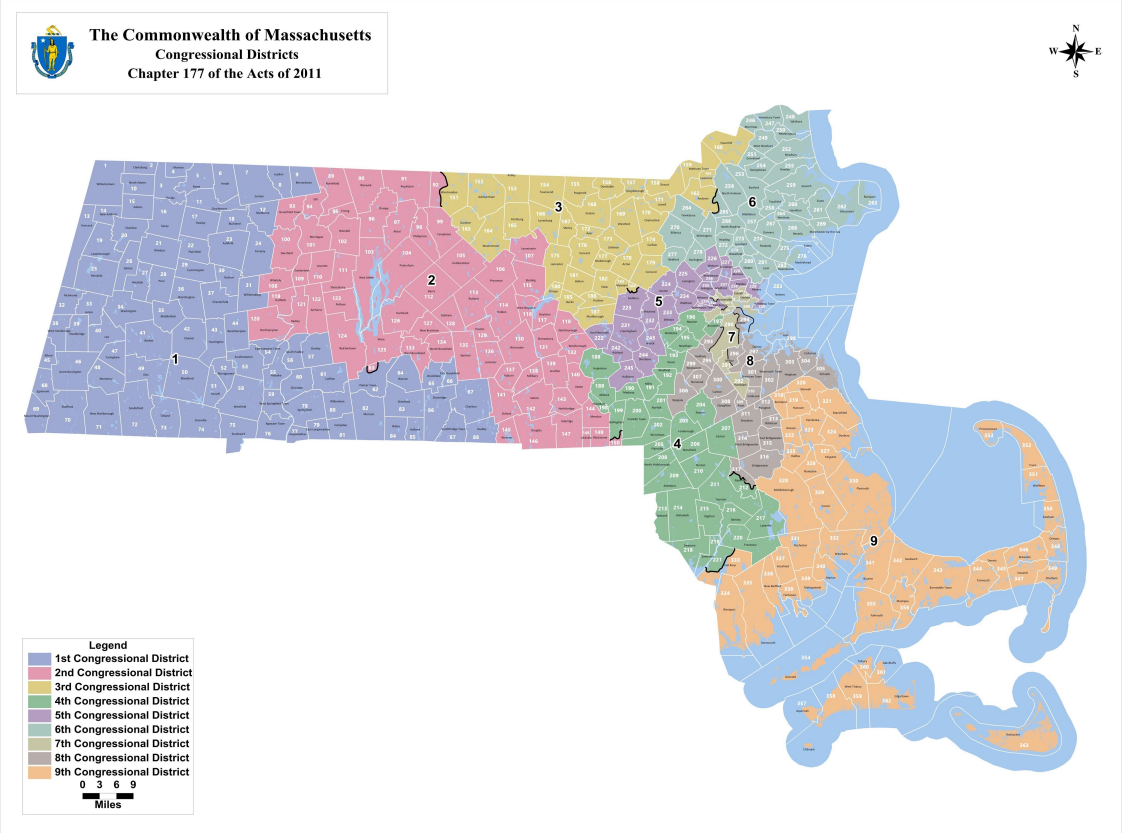


Figure 3.2: Labelled Massachusetts congressional district map

Some municipalities were also split between congressional districts, so it was necessary to divide them into two separate identifiers. For example, Winchendon, shown in figure 3.3 had to be split into two unique IDs, 92 and 151, 92 being in district 1 and 151 being in district 2. I labelled these in my data-set Winchendon2 and Winchendon3, the 2 and 3 signifying the district the ID belongs to.

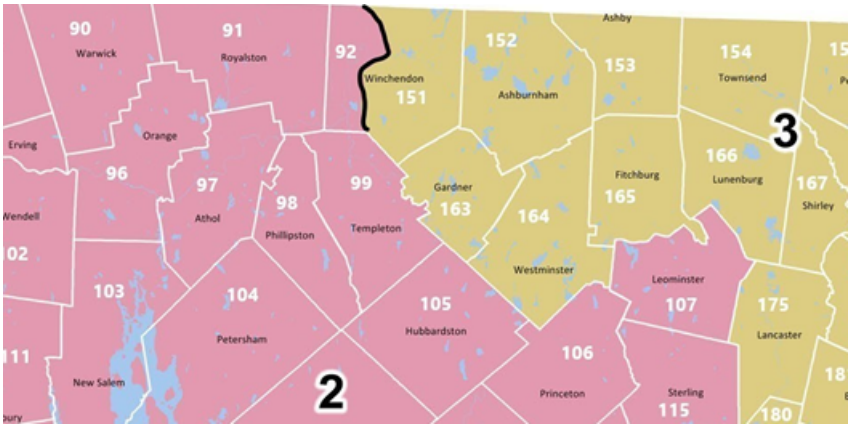


Figure 3.3: Winchendon's municipality split

Boston was split into four unique identifiers due to the fact that some parts of Boston weren't contiguous to other parts of Boston, even though the two parts resided in the same district. Figure 3.4 shows an example of this; ID 289 and 290 both belong to district 7 but are not contiguous, so had to be separated. During this process I also noted down which precincts and wards belonged to which district using detailed city maps. For example, ID 289 contains wards 1 and 2 of Boston. Figure 3.5 shows the detailed map of Boston used to collect this data.

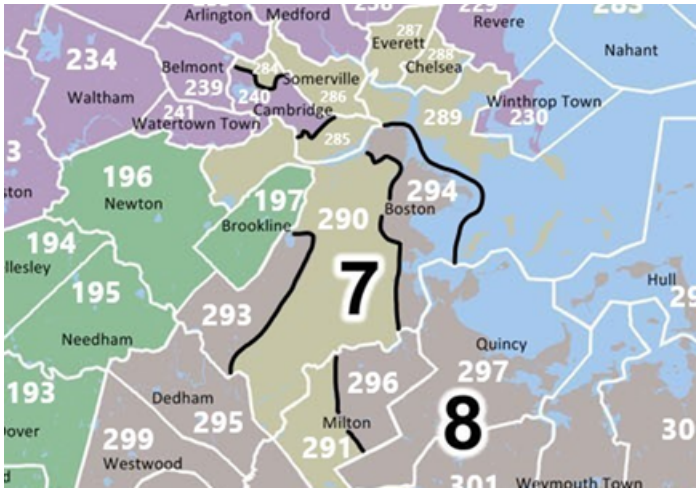


Figure 3.4: Boston's municipality split

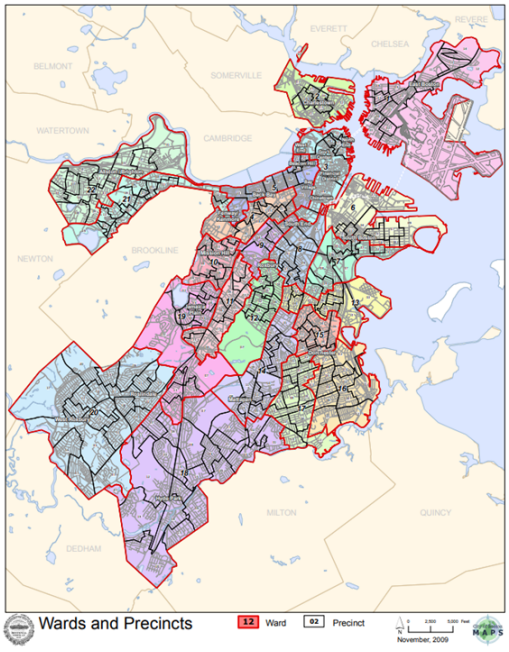


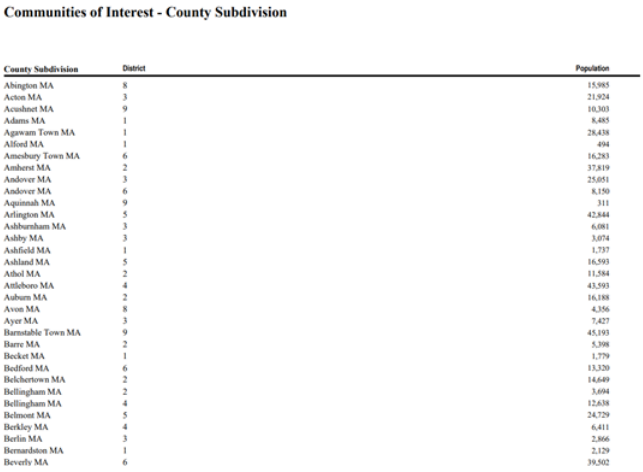
Figure 3.5: Boston's wards and precincts source

Having done rigorous checks to ensure that every municipality and split municipality was accounted for, I was left with the first step of my new data-set. Table 3.1 shows an excerpt of its current stage.

GlobalID	City/Town	District
1	Williamstown	1
2	Clarksburg	1
...	...	...
363	Nantucket	9

Table 3.1: Excerpt of first stage of data-set creation

The population of each municipality had to be added to the data-set next. Fortunately, Massachusett's government posted this information freely. The government PDF is shown in figure 3.6.



Communities of Interest - County Subdivision

County Subdivision	District	Population
Abington MA	8	15,985
Acton MA	3	21,924
Acushnet MA	9	10,303
Adams MA	1	8,485
Agawam Town MA	1	28,438
Alford MA	1	494
Amesbury Town MA	6	16,283
Amherst MA	2	37,819
Andover MA	3	25,051
Andover MA	6	8,150
Aquinnah MA	9	311
Arlington MA	5	42,844
Ashburnham MA	3	6,081
Ashby MA	3	3,074
Ashfield MA	1	1,737
Ashland MA	5	16,593
Athol MA	2	11,584
Attleboro MA	4	43,593
Auburn MA	2	16,188
Avon MA	8	4,356
Ayer MA	3	7,427
Barnstable Town MA	9	45,193
Barre MA	2	5,398
Becket MA	1	1,779
Bedford MA	6	13,320
Belchertown MA	2	14,649
Bellingham MA	2	3,694
Bellingham MA	4	12,638
Belmont MA	5	24,729
Berkley MA	4	6,411
Berlin MA	3	2,866
Bernardston MA	1	2,129
Beverly MA	6	39,502

Figure 3.6: Populations of municipalities data source

I attempted implementing a PDF scanning script in python, but I soon realised that it would take the same amount if not more time than manually copying over the populations. The task of copying populations had to be done meticulously. I verified that I copied the numbers over correctly by summing the copied numbers for each district and seeing if it matched with the sums given by the data source.

There were two different sources I needed for the voting data. One source contained the voting data at the precinct level, and one contained the data at the municipality level. These sources can be seen in the tables below.

City/Town	Joseph R. Biden, Jr.	Donald J. Trump
Abington	5,209	4,236
Acton	11,105	2,471
...	...	...
Yarmouth	9,149	5,993

Table 3.2: Excerpt of voting data source 1

City/Town	Ward	Pct	Joseph R. Biden, Jr.	Donald J. Trump
Abington	-	1	1,037	764
Abington	-	2	960	838
Abington	-	3	954	848
...	...	...	...	...
Yarmouth	-	7	1,596	1,059

Table 3.3: Excerpt of voting data source 2

It is worth noting that the original source contained information for other political parties but due to the nature of the two-party system in the US, this data was unnecessary to keep.

For all the municipalities which weren't split, I used python to merge the voting data with the data-set I created. The code for this can be seen in Appendix section 1. It performed a merge based on the name of the municipality and printed any rows that didn't get merged.

Finally, I labelled the adjacencies for each municipality. This had to be done extremely carefully because a single mistake could give misleading results when applying MCMC. I also labelled them both ways: for example, if ID 1 is neighbouring ID 2, I put 2 into 1's adjacency list, and I put 1 into 2's adjacency list. After labelling each municipalities' adjacent municipalities, I performed verification by using a graph package in Python. I created a directed graph using the adjacency data and checked for any vertices which had a one-way relationship with another. If this relationship existed, that indicated that there may be an error. The code used for this is shown in Appendix section 2.

My data-set was then complete. An excerpt of the final data set can be seen in table 3.4. My data-set can be accessed through the following link: https://github.com/amarkhullar/Gerrymandering

GlobalID	City/Town	District	Population	Adjacencies	Joseph R. Biden, Jr.	Donald J. Trump
1	Williamstown	1	7754	2,10,15,13,14	3,196	499
2	Clarksburg	1	1702	1,10,3	583	390
363	Nantucket	9	10172	362	5,241	1,914

Table 3.4: Excerpt of final data-set

3.2 Implementing Markov Chain Monte Carlo

To implement MCMC in Python I used a library called networkx to help with graph functions. I made a function called MakeGraph to read in my data-set into a graph format. This function can be seen in Appendix section 3. It creates a graph where each vertex represents one ID from my data-set. These vertices store data about the municipality it is representing, for example population, district, vote data. The function also adds edges between vertices which are adjacent.

I created a number of helper functions both before and during implementation of MCMC as these would be reused often. Some of the functions and their uses are below:

- GetNumVotes() - Returns the total number of votes for each party
- CalculateSeatsWon() - Returns the number of seats won by each party
- GetDistrict() - Returns a sub-graph of the map containing only the district number passed to it
- GetNeighbours() - Returns the nodes neighbouring a specific node
- GetOppositeNeighbour() - Returns the neighbour of a node that resides in another district (if it exists)
- GetPopulation() - Returns the population of a district
- GetRandomDistrict() - Samples a district proportionally to the size of the districts (For example, district 1 will occur more often because it contains more nodes)

3.2.1 Efficiency gap

The efficiency gap function is one of the key functions in my program. It calculates the efficiency gap of a map by iterating through the districts and calculating wasted votes. It is a relatively simple function so I will omit any code explanation.

3.2.2 Boundary flip

My boundary flip implementation, in Appendix section 3, is also quite simple. It iterates a specified number of rounds, and at each iteration it samples a random district, samples a random boundary node from said district, and randomly flips either the node or the node's neighbour with a 50/50 chance. If this flip satisfies the acceptance function, it is accepted into the graph.

The acceptance function initially takes two things into consideration, population balance and compactness. Population balance is kept in check by allowing a 5% deviation from the original average district population. This provides a good middle-ground between being stringent and faster convergence. Compactness is embedded into the acceptance function by using a ratio of number of nodes and diameter for each district's sub-graph. The diameter of a graph is the 'greastest shortest-path' in a graph (Weisstein 2021). Consider all the shortest paths from every node to every other node, the diameter is the longest path of this set of short paths.

By putting compactness into my acceptance function, the time spent per accepted flip rose. However, this was a worthwhile trade-off because without it, the sampled maps were long winding and had poor compact ratio scores.

While implementing boundary flip I realized that contiguity was not always being maintained. This is possible with the boundary flip proposal, so I implemented a contiguity constraint into the acceptance function. This works by simply checking that every district's sub-graph is fully-connected.

3.2.3 Recombination of districts

Implementing recombination of districts required more tuning than boundary flip. It starts in a similar approach to boundary flip, by sampling a random district. Then it samples a boundary node of that district and stores the district which neighbours it. After creating the induced subgraph, the algorithm works as follows:

```
while cuttable is False:

if numTries > tryLimit:
    break

spanningTree = self.GetSpanningTree(subgraph)
numTries += 1

for edge in spanningTree.edges():

    if self.IsAcceptedCut(graph,spanningTree,edge):

        cuttable = True
        newGraph = self.UpdateGraph(graph,edge,spanningTree,districtNo,neighbourDistrictNo)
        return newGraph, cuttable
```

The function iterates until the graph is either able to be validly cut, or if the try limit is reached. At each step of the iteration, a random spanning tree is generated. This works by setting the weights of the edges in the spanning tree to a random number sampled from a uniform distribution. After setting the weights, the maximum spanning tree is taken. This results in a random spanning tree due to the sampling of the weights. The code for this can be seen in GetSpanningTree in section 3 of the Appendix.

After the spanning tree is generated, we iterate through the edges of the spanning tree. At each edge, we check to see if cutting the tree results in a cut that satisfies our acceptance function. If it does, we update our graph and exit the function, otherwise we keep iterating the edges and other possible spanning trees.

Chapter 4

Results

I performed MCMC on the original map of Massachusetts using both proposals, boundary flip and recombination. I produced 10,000 samples from the boundary flip proposal and 500 samples from the recombination proposal. In all 10,000 boundary flip samples and 500 recombination samples, the Democrats won all 9 seats and the efficiency gap was 0.158. This is because in Massachusetts there is a strong Democrat majority, they hold 65% of the votes, and no possible district map that I sampled was able to produce a Republican seat, due to the relatively uniform spread of Republican votes in Massachusetts. With no Republican seat in any of the sampled maps, the number of wasted votes remains the same and the resulting data had no variation.

4.1 Original map statistics

To gain more insight, I extrapolated new data by modifying the voting data. Modifications were done by increasing the Republican vote share by a specific percentage, and setting the Democratic vote share to the remaining votes left. Code for this can be seen in the ChangeVotes function in section 3 in the Appendix. Table 4.1 shows some properties of the data I extrapolated, and the level by which they were modified.

Adjustment (%)	Vote share (D) (%)	Seats Won (D)	Vote share (R) (%)	Seats Won (R)	EG
0	67.1	9	32.9	0	0.158
15	62.2	9	37.8	0	0.256
25	58.9	8	41.1	1	0.201
35	55.6	7	44.4	2	0.165
45	52.3	4	47.7	5	0.106
55	49.0	2	51.0	7	0.278

Table 4.1: Political statistics of modified data on original map

Adjustment level is the percentage by which republican vote share was increased by, for example 15% means that the original republican vote count for each municipality was multiplied by 1.15 and rounded. Vote share is the percentage of votes for each party, out of only Republican

and Democrat. Seats won is the number of seats each party won on the original map. EG is the efficiency gap on the original map.

One observation of this data on the original map, it is noticeable that the number of seats won by the Republican party only increases slowly until the vote share is above 44.4%, and when it reaches 47.7% there is a large jump in Republican seats won, hence the efficiency goes down and we are closer to proportional representation. Another interesting point is that when the Republicans hold the majority vote share, with the 55% adjustment level, they win a disproportional number of seats and the efficiency gap rises drastically. However, this data gives us little to go on as we cannot compare these statistics against other maps.

4.1.1 Partisan symmetry

Figures 4.19 and 4.20 shows seats-votes curves I generated through extrapolating 300 data-sets, each with a 0.5% adjustment level increment. There is a curve from the Democratic point of view and the Republican point of view. The curve is interesting, it shows a partisan bias in favour of the Republicans, because when their vote share is at 50%, they hold a large majority of seats, far from a proportional number of seats. However, the drawback to this method is that the data is extrapolated, so results need to be taken with a pinch of salt.

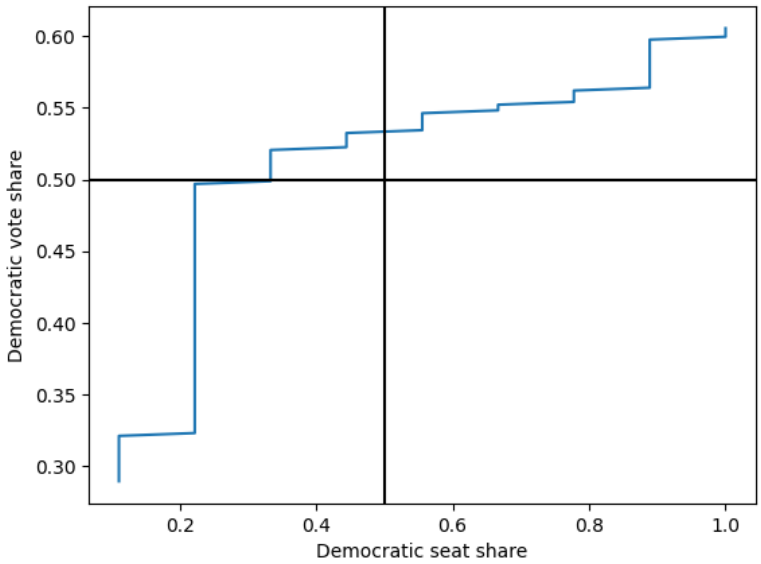


Figure 4.1: Seats-votes curve: Democratic view

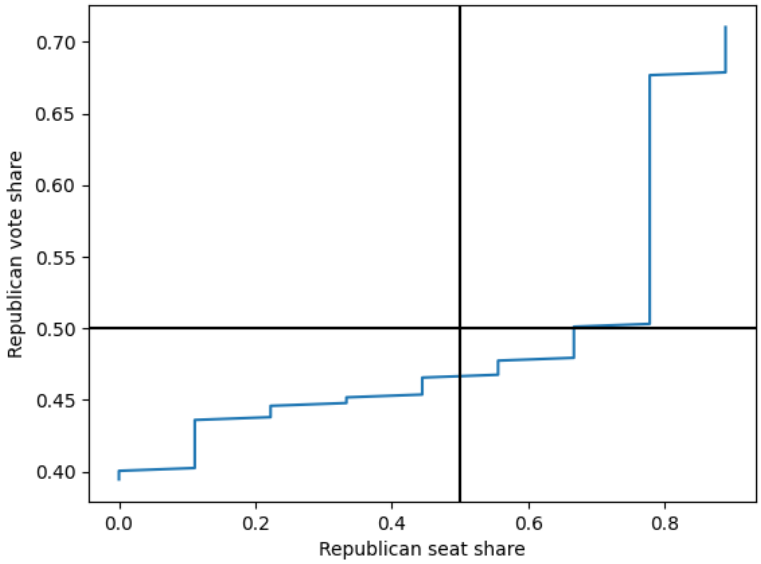


Figure 4.2: Seats-votes curve: Republican view

4.2 Ensemble sample figures

The figures below will show the results from MCMC of both proposals on all of the adjustment levels. The red lines indicate the score which the original map attained.

4.2.1 Adjustment level - 15%:

Boundary flip - All samples produced an efficiency gap of 0.256 and 0 Republican seats.

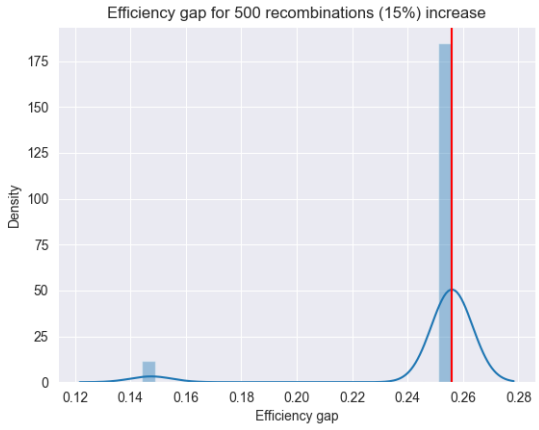


Figure 4.3: Efficiency gap histogram on 500 recombination samples (15%)

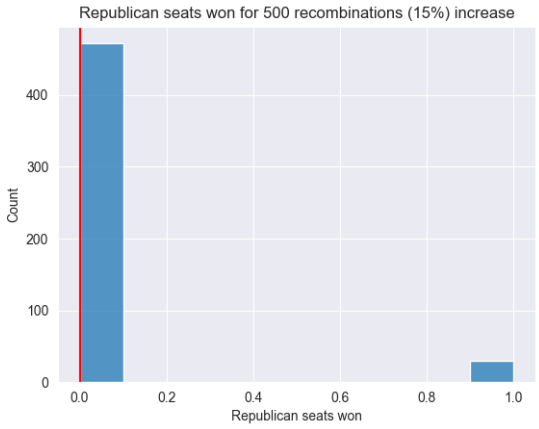


Figure 4.4: Republican seats won histogram on 500 recombination samples (15%)

4.2.2 Adjustment level - 25%:

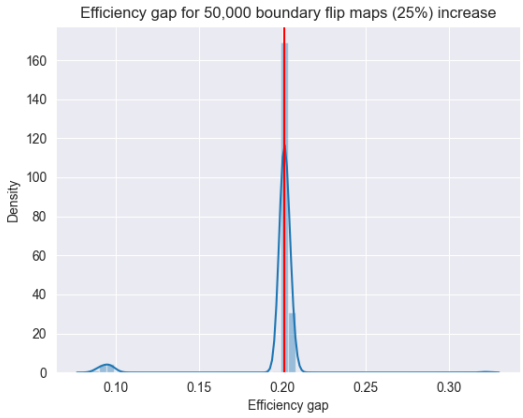


Figure 4.5: Efficiency gap histogram on 10,000 boundary flip samples (25%)

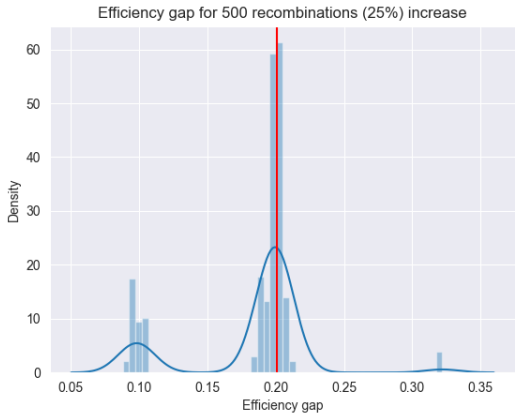


Figure 4.6: Efficiency gap histogram on 500 recombination samples (25%)

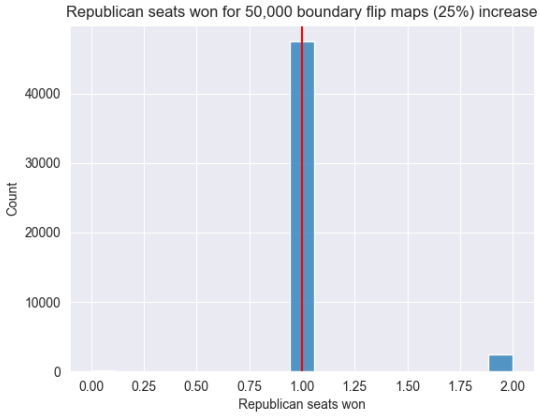


Figure 4.7: Republican seats won histogram on 10,000 boundary flip samples (25%)

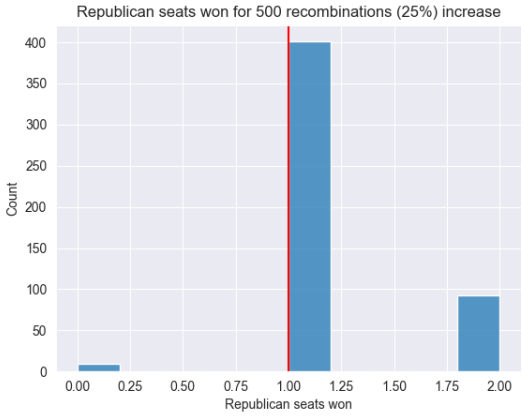


Figure 4.8: Republican seats won histogram on 500 recombination samples (25%)

4.2.3 Adjustment level - 35%:

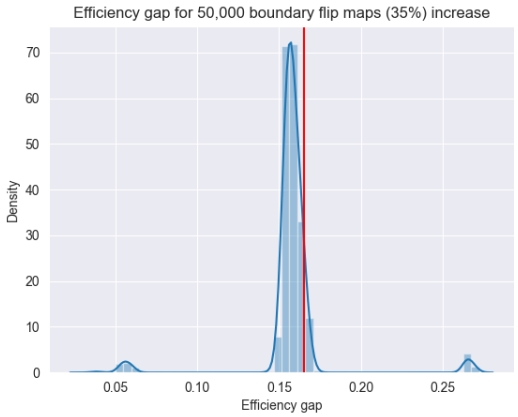


Figure 4.9: Efficiency gap histogram on 10,000 boundary flip samples (35%)

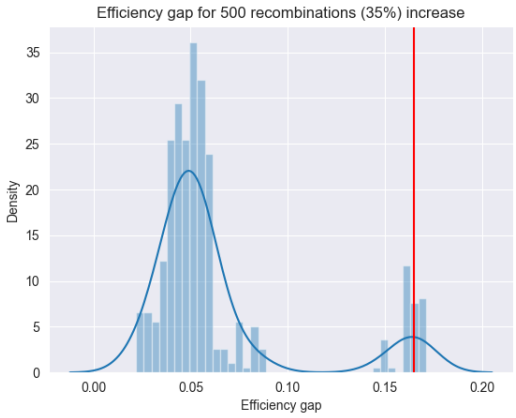


Figure 4.10: Efficiency gap histogram on 500 recombination samples (35%)

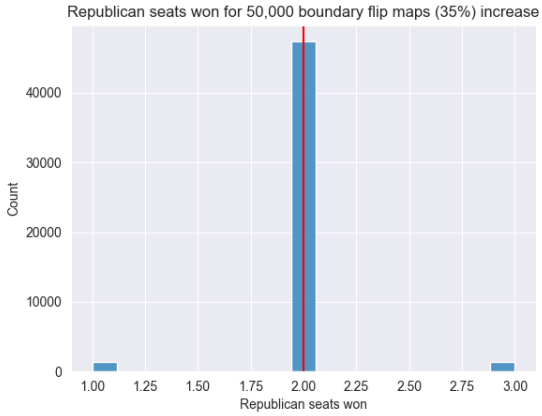


Figure 4.11: Republican seats won histogram on 10,000 boundary flip samples (35%)

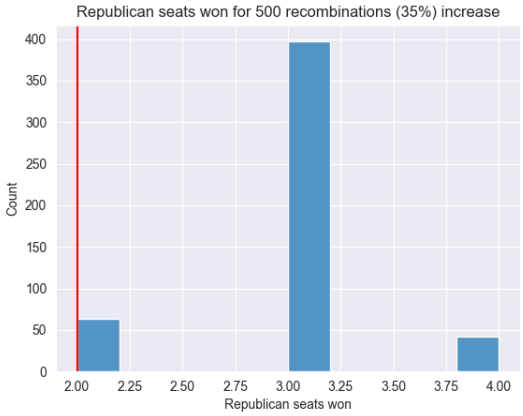


Figure 4.12: Republican seats won histogram on 500 recombination samples (35%)

4.2.4 Adjustment level - 45%:

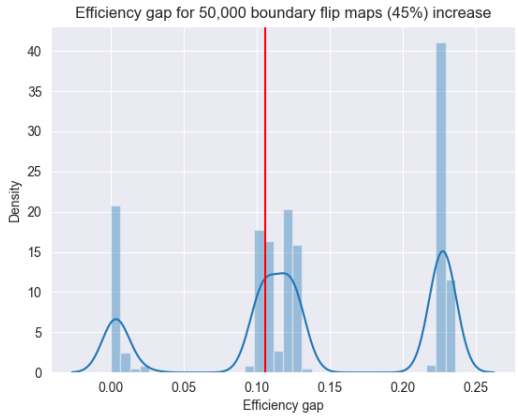


Figure 4.13: Efficiency gap histogram on 10,000 boundary flip samples (45%)

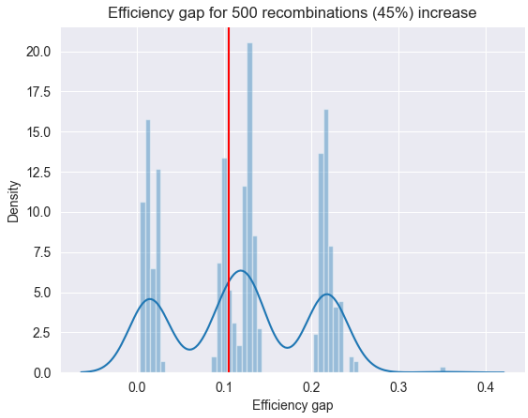


Figure 4.14: Efficiency gap histogram on 500 recombination samples (45%)

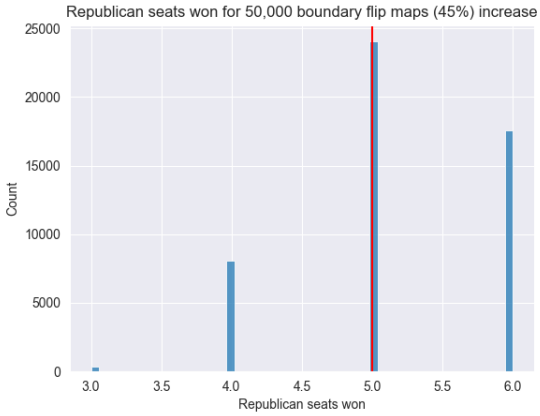


Figure 4.15: Republican seats won histogram on 10,000 boundary flip samples (45%)

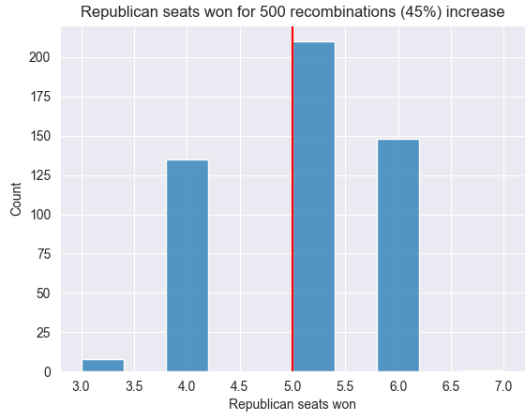


Figure 4.16: Republican seats won histogram on 500 recombination samples (45%)

4.2.5 Adjustment level - 55%:

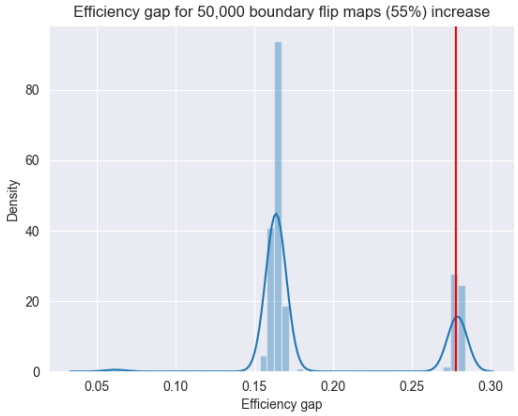


Figure 4.17: Efficiency gap histogram on 10,000 boundary flip samples (55%)

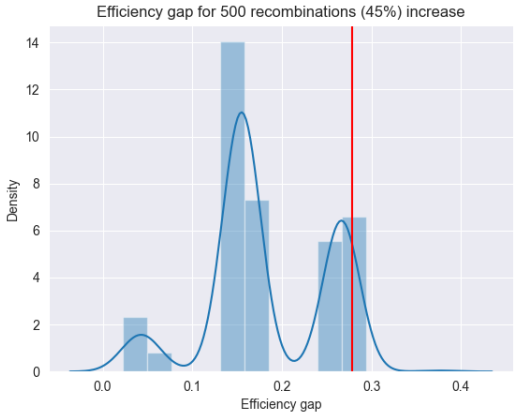


Figure 4.18: Efficiency gap histogram on 500 recombination samples (55%)

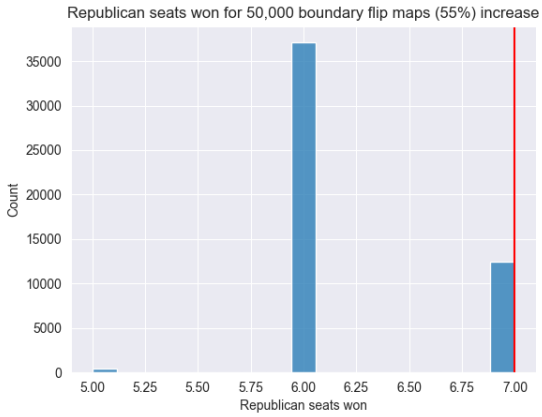


Figure 4.19: Republican seats won histogram on 10,000 boundary flip samples (55%)

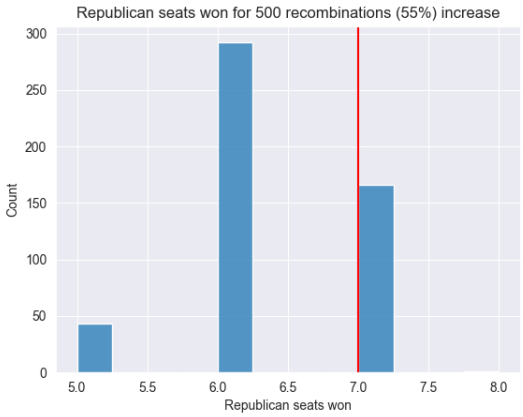


Figure 4.20: Republican seats won histogram on 500 recombination samples (55%)

4.3 Comparing the proposal functions

I will start by comparing the results from both proposals. Boundary flip and recombination both provided reasonable and similar results, however I believe the results show that recombination produced more diverse and higher quality samples. This is based upon the fact that at 15% adjustment, the recombination proposal was able to find maps that gained a Republican seat, while boundary flip didn't. As well as this, recombination found a wider range of Republican seats won in some of the adjustment levels (55%,45%).

However, in the 35% adjustment level in figures 4.7 and 4.8, it is noticeable how the variation in the samples differ. Boundary flip was able to find samples with a very high efficiency gap and 1 Republican seat which recombination didn't find. As well as this, the boundary flip method sampled more from the 0.14-0.16 efficiency gap range, while recombination sampled more results in the 0.05-0.08 efficiency gap range. Across the rest of the adjustment levels, the results were fairly similar.

I don't believe I came close to reaching the stationary distribution with either proposal. I suspect that many more samples are needed to draw conclusions on which proposal is better. Unfortunately, the run time of these algorithms prevented me from sampling in the quantity needed for this.

4.4 Analysing partisan bias

I assign more importance to the results from the adjustment levels closer to reality, because if partisan gerrymandering was happening, they would consider the possible near future and not a Republican majority envisaged in the 55% adjustment level.

I do not believe my results show any concrete evidence of democratic gerrymandering. In most of the figures the original map achieved near to the mean efficiency gap and seat share. The only exceptions are at the 35% adjustment level and the 55% adjustment level. At the 35% level, the Republicans won less seats than the mean and was a slight outlier, but not significant enough of an outlier to be called an extreme outlier. At the 55% level, the Republicans won more seats than the mean, but was not an outlier. I believe that more samples are needed to delve deeper into the space of possible maps, nonetheless the results I have show no evidence of partisan gerrymandering.

Chapter 5

Conclusions

5.1 Boundary flip vs Recombination

I do not believe that my experiments have determined which proposal function is objectively 'better'. However, I do think there are demonstrable positives and negatives. There may be some applications where one proposal function is more suited than the other.

Boundary flip has faster iterations, which allows you to sample results quicker. It also works well at sampling in the small neighbourhood around the original map. However, it's results seem to be less diverse and it takes longer to reach wider distributions.

Recombination is quick to spread out and diversify it's samples. It obtains a wide range of results in far fewer iterations than boundary flip, however each iteration takes a significantly longer amount of time than boundary flip so you cannot sample as many maps as quickly. Recombination is also better for larger graphs due to quicker convergence and more diversity.

5.2 Is Massachusetts gerrymandered?

The results from the 500 recombination samples on the 35% adjustment level do show that the original map contains some democratic bias, however this is the only level where the bias is significant. Furthermore, the data is extrapolated so it may not hold up well in a legal setting.

I have not seen any concrete evidence that Massachusetts has been gerrymandered. If I had more samples for each adjustment level, perhaps upwards of the 10,000 range for recombination and upwards of 1,000,000,000 for boundary flip, then more useful information may come to light. As well as this, results would have to show strong evidence of being an outlier to be useful. Strong bias would mean that the original map is more biased than at least 95% of other maps, and I don't believe that this strong bias can be satisfied with the current Massachusetts congressional district map.

5.3 Further Work

I believe that I have built a strong base for gerrymandering detecting, especially in Massachusetts because of the data-set that I created. Further work in the area could be to make the algorithms more efficient, and run more tests on the data-set to obtain enough samples to draw more concrete conclusions from. I could also use a greater density of adjustment levels with fewer gaps.

Nonetheless, the same code could be used to analyse partisan bias, or even racial bias, in other states, due to the generalized methodology of coding I used. One would simply need to convert to the same data format I used, or create a new MakeGraph function depending on the format of the input data.

Chapter 6

Appendix

6.1 Code used to merge data-sets

```
import pandas as pd

# read in own dataset
ids = pd.read_excel('nosplits.xls')
# read in voting dataset
votes = pd.read_csv('PD43+__2020_President_General_Election.csv')

# merge based on municipality name
result = pd.merge(ids,votes,on="City/Town")

# save resulting dataframe to excel format
result.to_excel('xx.xls')

# store any rows which didn't successfully merge
err = []
for idx,row in ids.iterrows():
    name = row['City/Town']
    if name not in result['City/Town'].values:
        err.append(name)
# print the rows which didn't merge
print(err)
```

6.2 Code used to verify adjacencies were correct

```
import pandas as pd
import matplotlib.pyplot as plt
import math
import random
import statistics
import numpy as np

# read in data
df = pd.read_excel('realfinaldata.xls')

# create adjacency matrix
adj = np.zeros((len(df)+1,len(df)+1))

# iterate through dataset
for row in df.iterrows():

    # retrieve node and node's adjacencies
    node = row[1]['GlobalID']
    adjacencies = str(row[1]['Adjacencies']).split(",")

    # for each adjacency, add it to the adjacency matrix
    for adjacency in adjacencies:

        intadj = int(adjacency)
        adj[node][intadj] = 1

# if there is a one way relationship between nodes, print it
for i in range(len(df)+1):
    for j in range(len(df)+1):
        if adj[i][j] != adj[j][i]:
            print(i,j)
```

6.3 MCMC implementation in Python

```
import networkx as nx
from networkx.algorithms import boundary, tree, distance_measures, components
import pandas as pd
import matplotlib.pyplot as plt
import math
import random
import statistics
import numpy as np
import csv

class MCMC():

    # initialization
    def __init__(self,dataFile):

        self.graph, self.noDistricts = self.MakeGraph(dataFile)
        self.graph, demTot, repTot = self.ChangeVotes(self.graph,55)
        print(repTot/(demTot+repTot))
        self.averageDistrictPopulation = self.CalculateAvgDistrictPop(self.graph)
        self.diameters = self.GetDistrictDiameters(self.graph)
        self.stats = []
        self.UpdateStats(self.graph)
        print(self.stats)

    # returns the total votes for democrats and republicans across the whole map
    def GetNumVotes(self,graph):

        demTot = 0
        repTot = 0

        for node in self.graph.nodes:

            demTot += graph.nodes[node]['dem']
            repTot += graph.nodes[node]['rep']

        return demTot,repTot

    # helper function
    def Printstuff(self):

        numRounds = 300000
        accepted = 0
        self.UpdateStats(self.graph)

        for i in range(numRounds):

            if accepted == 500:
                break
            newGraph,success = self.ReCom()
            if success:
                accepted += 1
                self.graph = newGraph
                self.UpdateStats(self.graph)
```

```
    print(accepted)

    plt.hist([stat["efficiency"] for stat in self.stats], bins="auto", alpha=0.5, facecolor='blue')
    plt.show()
    plt.hist([stat["dem"] for stat in self.stats], bins="auto", alpha=0.5, facecolor='blue')
    plt.show()
    plt.hist([stat["rep"] for stat in self.stats], bins="auto", alpha=0.5, facecolor='blue')
    plt.show()


    # save stats
    df = pd.DataFrame(self.stats)
    df.to_csv('recom100.csv')

# increases republican vote level by 'adjustmentLevel' percent
# sets the democratic vote as the remaining number of votes
def ChangeVotes(self,graph,adjustmentLevel):

    demTot = 0
    repTot = 0

    for node in graph.nodes:

        dem = graph.nodes[node]['dem']
        rep = graph.nodes[node]['rep']
        total = dem + rep

        # mu,sigma = adjustmentLevel,adjustmentLevel/5
        # epsilon = np.random.normal(mu,sigma,1)

        adjustmentPercentage = 1 + (adjustmentLevel/100)
        newRep = round(rep*adjustmentPercentage)
        newDem = total - newRep

        graph.nodes[node]['dem'] = newDem
        graph.nodes[node]['rep'] = newRep

        demTot += newDem
        repTot += newRep

    return graph, demTot,repTot

# Calculates the average population for each district
def CalculateAvgDistrictPop(self,graph):

    pops = []

    for i in range(1,self.noDistricts+1):

        district = self.GetDistrict(i,graph)
        pop = self.GetPopulation(district)
        pops.append(pop)

    return (sum(pops)/self.noDistricts)
```

```
# Reads in the excel file into a dual graph format suitable for MCMC
def MakeGraph(self,dataFile):

    df = pd.read_excel(dataFile)

    G = nx.Graph()
    tuples = []
    maxDistrict = 0

    # iterate through data
    for row in df.iterrows():

        id = int(row[1]['GlobalID'])

        # set node to datapoint
        G.add_node(id)
        # store relevant data in node
        G.nodes[id]['name'] = str(row[1]['City/Town'])
        G.nodes[id]['district'] = int(row[1]['District'])
        G.nodes[id]['pop'] = int((row[1]['Population']))
        G.nodes[id]['adj'] = row[1]['Adjacencies']
        G.nodes[id]['dem'] = int((row[1]['Joseph R. Biden, Jr.']))
        G.nodes[id]['rep'] = int((row[1]['Donald J. Trump']))
        G.nodes[id]['total'] = int(row[1]['Total Votes Cast'])

        adjacencies = str(row[1]['Adjacencies']).split(",")
        # add edges between adjacent precincts into tuple object
        for adjacency in adjacencies:
            tup = (id,int(adjacency))
            tuples.append(tup)

        # calculate the maximum district number (eg for MA it is 9 because there are 9 districts)
        if int(row[1]['District']) > maxDistrict:
            maxDistrict = int(row[1]['District'])

    # add edges into graph from tuple object
    G.add_edges_from(tuples)

    return G,maxDistrict

# calculates number of seats won for each party
def CalculateSeatsWon(self,graph):

    demSeats = 0
    repSeats = 0

    for district in range(1,self.noDistricts+1):

        # get current district and initialize vote tallies
        currentDistrict = self.GetDistrict(district,graph)
        dem = 0
        rep = 0

        # iterate through precincts in current district
        for node in currentDistrict:
```

```
            # tally current district votes
            dem += graph.nodes[node]['dem']
            rep += graph.nodes[node]['rep']

        if dem > rep:
            demSeats += 1
        else: # changed temporarily from elif rep > dem
            repSeats += 1

    return demSeats,repSeats

# Calculates the efficiency gap of the map
def EfficiencyGap(self,graph):

    demWasted = 0
    repWasted = 0
    totalVotes = 0
    # iterate through each district
    for district in range(1,self.noDistricts+1):

        # get current district and initialize vote tallies
        currentDistrict = self.GetDistrict(district,graph)
        dem = 0
        rep = 0

        # iterate through precincts in current district
        for node in currentDistrict:

            # tally current district votes
            dem += graph.nodes[node]['dem']
            rep += graph.nodes[node]['rep']

        votesNeeded = math.ceil((dem+rep)/2)
        totalVotes += dem+rep

        # add wasted votes to total
        if(rep > dem):
            demWasted += dem
            repWasted += (rep-votesNeeded)
        elif(dem > rep):
            repWasted += rep
            demWasted += (dem-votesNeeded)

    efficiency = abs(repWasted-demWasted)/totalVotes

    return round(efficiency,3)

# Returns a sub graph of the district which is passed to the function
def GetDistrict(self,districtNo,g):

    precincts =  [precinct for precinct in g.nodes if g.nodes[precinct]['district'] == districtNo]
    return g.subgraph(precincts)

# returns the neighbour nodes of a specified node
def GetNeighbours(self,node):
```

```
    return list(self.graph.adj[node])

# returns the other node at the boundary, the one in the other district
def GetOppositeNeighbour(self,node):

    currentDistrict = self.graph.nodes[node]['district']
    neighbours = self.GetNeighbours(node)

    for neighbour in neighbours:

        if self.graph.nodes[neighbour]['district'] != currentDistrict:
            return neighbour

# returns the population of a district
def GetPopulation(self,district):

    pop = 0

    for node in district:
        pop += district.nodes[node]['pop']

    return pop

# returns true if the graph is compact enough
def IsCompact(self,newGraph,oldGraph):

    threshold = 0.002
    newCompact = statistics.mean(self.GetCompactRatio(newGraph))
    oldCompact = statistics.mean(self.GetCompactRatio(oldGraph))
    difference = newCompact - oldCompact

    # we dont want the compact score to go up
    if difference > threshold:
        return False

    return True

# returns true if each district is contiguous (fully connected)
def IsContiguous(self,g):

    for districtNo in range(1,self.noDistricts+1):

        district = self.GetDistrict(districtNo,g)
        if not components.is_connected(district):
            return False

    return True

# returns true if the population is balanced within the threshold amount
def IsBalanced(self,g):

    # allowed population imbalance
    populationThreshold = 0.05
    satisfies = []

    for i in range(1,self.noDistricts+1):
```

```
        district = self.GetDistrict(i,g)
        pop = self.GetPopulation(district)

        if not (pop >= (self.averageDistrictPopulation*(1-populationThreshold))
        and pop <= (self.averageDistrictPopulation*(1+populationThreshold))):
            return False

    return True

# acceptance function
# based on population balance and contiguity// might include compactness later
def IsAccepted(self,newGraph,oldGraph):

    if self.IsBalanced(newGraph):
        if self.IsContiguous(newGraph):
            if self.IsCompact(newGraph,oldGraph):
                return True

    return False

# returns a list of boundary nodes for a specified district
def GetBoundaryNodes(self,district):

    return(boundary.node_boundary(self.graph, district.nodes))

# returns a list containing the graphical diameter for each district
# (diameter is the longest 'shortest-path' in each district)
def GetDistrictDiameters(self,graph):

    diameters = []

    for districtNo in range(1,self.noDistricts+1):

        district = self.GetDistrict(districtNo,graph)
        diameter = distance_measures.diameter(district)
        diameters.append(diameter)

    return diameters

# returns a list containing the total number of nodes in each district
def GetNumDistrictNodes(self,graph):

    nodes = []

    for districtNo in range(1,self.noDistricts+1):

        district = self.GetDistrict(districtNo,graph)
        nodes.append(len(district))

    return nodes

# returns a list of compactness ratios of diameter / numNodes , one value for each district
def GetCompactRatio(self,graph):

    numNodes = self.GetNumDistrictNodes(graph)
```

```
    diameters = self.GetDistrictDiameters(graph)
    ratio = [i / j for i, j in zip(diameters, numNodes)]
    return ratio

# updates statistical data
def UpdateStats(self,graph):

    newStats = {}

    newStats['efficiency'] = self.EfficiencyGap(graph)

    demSeats,repSeats = self.CalculateSeatsWon(graph)
    newStats['dem'] = demSeats
    newStats['rep'] = repSeats

    self.stats.append(newStats)

# performs boundary flip algorithm
def BoundaryFlip(self,numRounds): # add numRounds

    accepted = 0
    print(self.stats)

    for i in range(numRounds):

        if accepted == 50000:
            return accepted
        # generate random district to perform a boundary flip on
        # randomly proportional to number of nodes, not random between districts
        numNodes = len(list(self.graph.nodes))
        randomNode = random.randint(1,numNodes)
        districtNo = self.graph.nodes[randomNode]['district']

        # get district subgraph
        district = self.GetDistrict(districtNo,self.graph)
        # get all boundary nodes of random district
        boundaryNodes = self.GetBoundaryNodes(district)
        # sample a random boundary node
        boundaryNode = random.sample(boundaryNodes, 1)[0]

        # get node from other district at the boundary
        oppositeNode = self.GetOppositeNeighbour(boundaryNode)

        g = self.graph.copy()

        # 50/50 between flipping original or opposite node
        if random.choice([0,1]) == 0:
            g.nodes[boundaryNode]['district'] = g.nodes[oppositeNode]['district']
            #print(boundaryNode,'to',g.nodes[oppositeNode]['district'])
        else:
            g.nodes[oppositeNode]['district'] = g.nodes[boundaryNode]['district']
            #print(oppositeNode,'to',g.nodes[boundaryNode]['district'])

        if self.IsAccepted(g,self.graph):
            accepted += 1
            self.graph = g
```

```
            self.UpdateStats(self.graph)

    return accepted

# returns a random district proportional to the number of nodes in the graph
# (we want to sample according to district size)
def GetRandomDistrict(self,graph):

    numNodes = len(graph.nodes)
    randomNode = random.randint(1,numNodes)
    districtNo = graph.nodes[randomNode]['district']
    district = self.GetDistrict(districtNo,graph)
    return district,districtNo

# returns a random spanning tree of a graph
def GetSpanningTree(self,graph):

    rand = np.random.uniform(0, 1, len(graph.edges))

    i = 0
    for edge in graph.edges:
        graph.edges[edge]["weight"] = rand[i]
        i += 1

    spanningTree = tree.maximum_spanning_tree(
        graph, weight="weight"
    )

    return spanningTree

# get population of an entire graph
def SumGraphPop(self,graph):

    total = 0
    for node in graph:
        total += graph.nodes[node]['pop']
    return total

# returns true if the graph cut passed to it conforms to population balance
def IsAcceptedCut(self,graph,spanningTree,edge):

    populationThreshold = 0.05

    (u,v) = edge
    tree = spanningTree.copy()
    tree.remove_edge(u,v)

    newGraphs = list(nx.connected_components(tree))
    g1 = graph.subgraph(newGraphs[0])
    g2 = graph.subgraph(newGraphs[1])

    pop1 = self.SumGraphPop(g1)
    pop2 = self.SumGraphPop(g2)

    if not (pop1 >= (self.averageDistrictPopulation*(1-populationThreshold))
    and pop1 <= (self.averageDistrictPopulation*(1+populationThreshold))):
```

```
            return False
        if not (pop2 >= (self.averageDistrictPopulation*(1-populationThreshold))
        and pop2 <= (self.averageDistrictPopulation*(1+populationThreshold))):
            return False

        return True

    # returns updated graph after successful ReCom cut
    def UpdateGraph(self,graph,edge,spanningTree,district1,district2):

        (u,v) = edge
        tree = spanningTree.copy()
        tree.remove_edge(u,v)

        newGraphs = list(nx.connected_components(tree))
        g1 = graph.subgraph(newGraphs[0])
        g2 = graph.subgraph(newGraphs[1])

        for node in g1.nodes:
            graph.nodes[node]['district'] = district1
        for node in g2.nodes:
            graph.nodes[node]['district'] = district2

        return graph

    # performs one iteration of the ReCom algorithm (Deford,Duchin,2020)
    def ReCom(self):

        graph = self.graph.copy()
        # get a random district
        randomDistrict,districtNo = self.GetRandomDistrict(graph)

        # get boundary nodes for said district
        boundaryNodes = boundary.node_boundary(graph, randomDistrict.nodes)
        # sample a random boundary node
        boundaryNode = random.sample(boundaryNodes, 1)[0]

        # get neighbour district
        neighbourNode = self.GetOppositeNeighbour(boundaryNode)
        neighbourDistrictNo = graph.nodes[neighbourNode]['district']
        neighbourDistrict = self.GetDistrict(neighbourDistrictNo,graph)

        # create induced subgraph, l = 2
        subgraph = graph.subgraph(list(randomDistrict.nodes) + list(neighbourDistrict.nodes))

        # perform ReCom algorithm
        cuttable = False
        numTries = 0

        numNodes = len(subgraph.nodes)
        # adjust the limit depending on the number of nodes in the graph
        tryLimit = numNodes

        while cuttable is False:

            if numTries > tryLimit:
```

```
                break

            spanningTree = self.GetSpanningTree(subgraph)
            numTries += 1

            for edge in spanningTree.edges():

                if self.IsAcceptedCut(graph,spanningTree,edge):

                    cuttable = True
                    newGraph = self.UpdateGraph(graph,edge,spanningTree,districtNo,neighbourDistrictNo)
                    return newGraph, cuttable

        return None,cuttable


newMCMC = MCMC('realfinaldata.xls')
newMCMC.Printstuff()
```

Chapter 7

Sources

7.1 Bibliography

Bibliography

[DeF19] Daryl DeFord. “Introduction to Discrete MCMC for Redistricting”. In: (2019). DOI: https://people.csail.mit.edu/ddeford/MCMC_Intro_plus.pdf.

[DeF20] Daryl DeFord. “Recombination: A family of Markov chains for redistricting”. In: (2020). DOI: https://mggg.org/uploads/ReCom.pdf.

[Dub16] Matthew Dube. “Beyond the circle: Measuring district compactness using graph theory”. In: (2016). DOI: https://www.researchgate.net/publication/311557290_Beyond_the_Circle_Measuring_District_Compactness_using_Graph_Theory.

[Duc18a] Moon Duchin. “Geometry vs Gerrymandering”. In: (2018). DOI: https://www.scientificamerican.com/article/geometry-versus-gerrymandering/.

[Duc18b] Moon Duchin. *Redistricting and Representation*. 2018. URL: https://www.amacad.org/news/redistricting-and-representation.

[Ell20] Jordan S. Ellenberg. “GEOMETRY, INFERENCE, COMPLEXITY, AND DEMOCRACY”. In: (2020). DOI: https://arxiv.org/abs/2006.10879.

[Fif20] Benjamin Fifield. “Automated Redistricting Simulation Using Markov Chain Monte Carlo”. In: (2020). DOI: https://imai.fas.harvard.edu/research/files/redist.pdf.

[Kas98] Robert E. Kass. “Markov Chain Monte Carlo in Practice: A Roundtable Discussion”. In: (1998). DOI: http://www.stat.cmu.edu/~kass/papers/MCMCinPractice1998.pdf.

[Lap18] Issie Lapowski. *The Geeks Who Put a Stop to Pennsylvania’s Partisan Gerrymandering*. 2018. URL: https://www.wired.com/story/pennsylvania-partisan-gerrymandering-experts/.

[Leg21] Massachusetts Legislature. *Massachusetts constitution*. 2021. URL: https://malegislature.gov/Laws/Constitution#amendmentArticleXVI.

[Wei20] Eric W. Weisstein. *Graph cycle*. 2020. URL: https://mathworld.wolfram.com/GraphCycle.html.

[Wei21] Eric W. Weisstein. *Graph diameter*. 2021. URL: https://mathworld.wolfram.com/GraphDiameter.html.

7.2 Figure sources

1.1 Massachusetts' congressional district resembling a salamander:

https://en.wikipedia.org/wiki/Gerrymandering#/media/File:The_Gerry-Mander_Edit.png

1.2 How to steal an election:

https://commons.wikimedia.org/wiki/File:How_to_Steal_an_Election_-_Gerrymandering.svg

2.1 Example Markov Chain:

https://towardsdatascience.com/markov-chain-analysis-and-simulation-using-python-4507cee0b06e

2.2 Dual graph of Iowa:
(Duchin 2018a)
2.6 Example graph with cycles

https://en.wikipedia.org/wiki/Cycle_(graph_theory)#/media/File:Graph_cycle.svg

2.13 Seats-votes curves for Minnesota and Ohio 2016

https://www.amacad.org/sites/default/files/academy/images/publications
/bulletin/winter2018/bulletin_Winter2018_Redistricting-Fig2.jpg

3.1 Massachusetts congressional district map

https://malegislature.gov/StateHouse/MediaGallery/Image/Proposed%20Congressional%20Districts.jpg

3.5 Boston's wards and precincts source

https://www.cityofboston.gov/maps/pdfs/ward_and_precincts.pdf