Markov Chain Monte Carlo: A Tool for Gerrymandering Analysis

Rishabh Sinha

September 18, 2024

1 Introduction

The goal of this project is to create a version of a successful Markov Chain Monte Carlo (MCMC) gerrymandering algorithm and showcase its effectiveness. To achieve this, we will follow these steps:

1. Understand the MCMC algorithm by applying it to a different problem, which will provide deeper insight for its application to gerrymandering.
2. Study the history of gerrymandering, and past methods used to combat it, along with their advantages and shortcomings.
3. Analyze the MCMC algorithm for gerrymandering, comprehend its intricacies, and write code to demonstrate its effectiveness.

By following these steps, we establish MCMC as an advantageous method for calculating gerrymandering, offering insight into both gerrymandering's historical context and the Monte Carlo algorithm's depth.

2 Code-Breaking using MCMC

The Problem : We are presented with a scrambled sequence of n characters, which we need to decipher to an english sentence. Our only clue is that each character in the message has been substituted with a different character.
Our Objective: To decrypt the original message.

To tackle the problem of decrypting a scrambled message, we will start with a simpler example. Consider the encrypted code *hamt*, which should decrypt to *math*. For simplicity and scalability, we will operate in a closed system where only the letters a, h, m, t exist.

Given that there are only 4 letters, there are 4! permutations possible. For instance, m can be substituted by any of the 4 letters a, h, m, t. Consequently, a would need to be substituted by one of the remaining 3 letters, t by one of the remaining 2, and h by the last one.

Since $4! = 24$, the number of permutations is small enough to manually compute by examining each possible substitution to determine the decrypted word. However, as the number of characters increases, this method becomes impractical.

Including spaces, the total number of possible substitutions for all the letters of the alphabet is

$$27! = 10,888,869,450,418,352,160,768,000,000 \approx 10^{19}$$

This astronomical number makes it impossible to compute manually, necessitating the development of alternative methods to decrypt larger codes.

Given that our final decrypted message should form valid English words, we can leverage this knowledge to determine the likelihood of certain substitutions. We achieve this by breaking down our code into *bigrams*, which are pairs of consecutive characters. By assessing the frequency of these bigrams in the English language, we can estimate how likely they are to appear.

For our earlier example, the encrypted code *hamt* would produce the following bigrams:

$$ha, am, mt$$

To evaluate the 'likelihood' or 'probability' of a bigram's occurrence, we require a large dataset of English text and well-defined rules to minimize bias. One suitable source for such data is the book *War and Peace* by *Leo Tolstoy.* While it doesn't have to be this specific book, it must be an extensive English text to ensure a representative sample.

War and Peace contains $587,287$ words, making it a suitable choice for extracting bigram data.

Each bigram will be assigned a probability based on its frequency in our dataset. For instance, if a text contains 1000 characters, resulting in 999 bigrams, and the bigram $'ha'$ appears 20 times, the probability is calculated as:

$$p(ha) = \frac{20}{999} \approx 0.0202$$

Using this system, we can calculate the plausibility of each code P using the formula:

$$P = \prod_i p(x_i, x_{i+1})$$

where x_i is the i^{th} character in the encoded message.

For example, the plausibility of *hamt* would be:

$$P(hamt) = p(ha) \cdot p(am) \cdot p(mt)$$

Our expectation is that the code forming coherent English words will have the highest plausibility value. We can utilize an algorithm based on this plausibility value to decrypt the encoded message.

The algorithm we use consists of the following steps:

1. Begin with a substitution (π_0) chosen at random.
2. Given π_n, form a new substitution $\pi' = \sigma\pi_n$, where σ is a transposition that randomly swaps any two characters.
3.
    - If $P(\pi') \geq P(\pi_n)$, then $\pi' = \pi_{n+1}$.
    - If $P(\pi') < P(\pi_n)$, then $P(\pi')/P(\pi_n) \leq 1$. Let $P(\pi')/P(\pi_n) = x$.
      Let $y \sim P[0,1]$.
      If $y < x$, then $\pi' = \pi_{n+1}$;
      else, we discard π' and return to Step 2.
4. Repeat for n iterations.

This concept is intuitively easy to understand: at each step, we generate a new substitution and accept it if it is more plausible. However, there is a chance that a less plausible version will be accepted, depending on the difference in plausibility between the two versions.

In essence, rather than solely searching for the most plausible substitution, the algorithm samples from the substitutions in such a way that the more plausible permutations appear more frequently. This approach is known as the Markov Chain Monte Carlo and utilizing the Metropolis-Hastings Algorithm.

The Python code implementing the algorithm is available in the attached GitHub repository.

3 The Metropolis - Hastings Algorithm

The Metropolis-Hastings algorithm is an MCMC method that helps sample from complex distributions. It works by suggesting new points based on the current state and then deciding whether to accept them based on a probability that ensures the sample accurately reflects the target distribution.

The Algorithm is as follows :

1. Initialize $x^{(0)}$
2. For $i = 0$ till $N - 1$
    - Sample $u \sim U_{[0,1]}$
    - Sample $x^* \sim q(x^*|x^{(i)})$
    - If $u < A(x^*, x^{(i)}) = min\left(1, \frac{p(x^*)q(x^*|x^{(i)})}{p(x^{(i)})q(x^{(i)}|x^*)}\right)$
      $x^{(i+1)} = x^*$ else
      $x^{(i+1)} = x^{(i)}$

The Metropolis-Hastings algorithm relies on the acceptance probability function A, which depends on two functions: q and p.

The function q represents the proposal distribution, $q(x^* \mid x^{(i)})$, which determines how likely it is to propose state x^* given the current state $x^{(i)}$. By choosing q appropriately, we can influence the sampling process. For instance, a higher q score for certain states means these states are more likely to be proposed from other states.

In cases where the algorithm assumes a symmetric proposal distribution (i.e., $q(x \mid y) = q(y \mid x)$), this symmetry cancels out in the acceptance probability calculation. Consequently, the acceptance probability depends only on p.

The function p represents the target distribution, which gives the probability of each state. While p does not need to be the exact probability, it should be proportional to it. This proportionality ensures that the sampling reflects the target distribution, with states occurring in proportion to their likelihood.

Although the algorithm itself is straightforward, finding suitable functions p and q can be challenging and complex.

4 GerryMandering

What is Gerrymandering?
Gerrymandering is the political manipulation of electoral district boundaries to create an unfair advantage, allowing one party to gain more seats and power within a state.

Over the years, various methods have been developed to detect and measure gerrymandering. The three key approaches we'll explore are:

1. **Partisan Symmetry**: Assesses whether each party would receive the same number of seats for the same percentage of votes, aiming for fairness in representation.

2. **Efficiency Gap**: Measures the difference between the parties' wasted votes—those beyond what was needed to win—highlighting any unfair advantage.

3. **Markov Chain Monte Carlo (MCMC)**: Generates random districting plans to compare against actual plans, identifying potential gerrymandering through statistical simulations.

Partisian Symmetry

Partisan symmetry is a concept in electoral fairness that originated from the study of gerrymandering in the 1980s. Political scientists developed it to assess whether electoral systems treat political parties equally. The core idea is that if one party receives a certain percentage of votes and secures a specific number of seats, the other party should receive the same number of seats with an identical vote percentage. This concept helps ensure that the translation of votes into seats is fair.

Various methods exist to calculate partisan symmetry, but we will focus on one of the earliest models developed by Gary King in 1987.

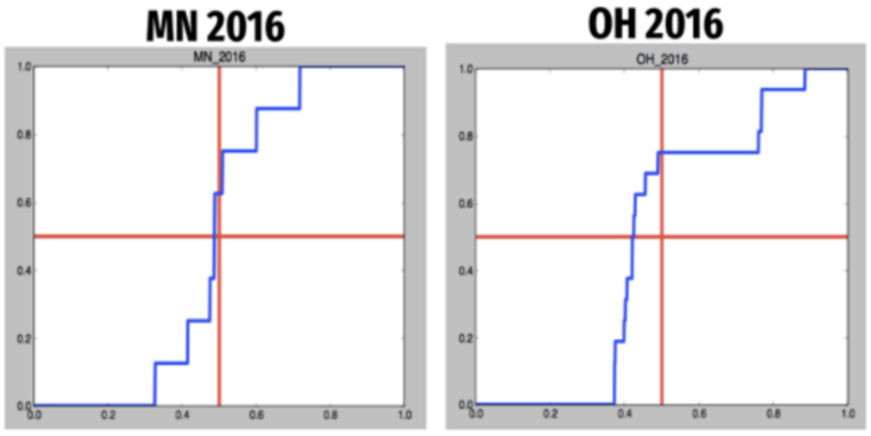

Figure 1: An example of a Republican Seats/Votes graph from 2016 for Ohio and Minnesota, showcasing partisan symmetry and potential gerrymandering. Seats are on the Y-axis, and votes are on the X-axis.

To assess whether gerrymandering is occurring using partisan symmetry, the primary approach is to create a Seats/Votes graph for a party. An example is given in 1.

From 1, we can observe deviations from symmetry. For example, in Ohio 2016, with just around 40% of the votes, a party was able to secure a majority seat share, suggesting potential gerrymandering.

There are multiple ways to graph the Seats/Votes relationship. The above example was constructed by going district by district, adding proportional seats and votes for a party across the state, and plotting them on a step graph. However, our focus will be on modeling this graph using a function.

We model this graph using the cube law function:

$$\left(\frac{S}{1-S}\right) = \left(\frac{V}{1-V}\right)^{\rho}$$

where S is seats, V is votes, and ρ is an undetermined parameter.

The value of ρ reflects the type of representation in the political system. Figure 2 provides further details on how different values of ρ affect the graph.

4.1 Incorporating Bias

The model described above reflects political representation without bias. To incorporate bias towards a specific party, we modify the equation as follows:

$$\left(\frac{S}{1-S}\right) = \beta\left(\frac{V}{1-V}\right)^{\rho}$$

where β represents bias.

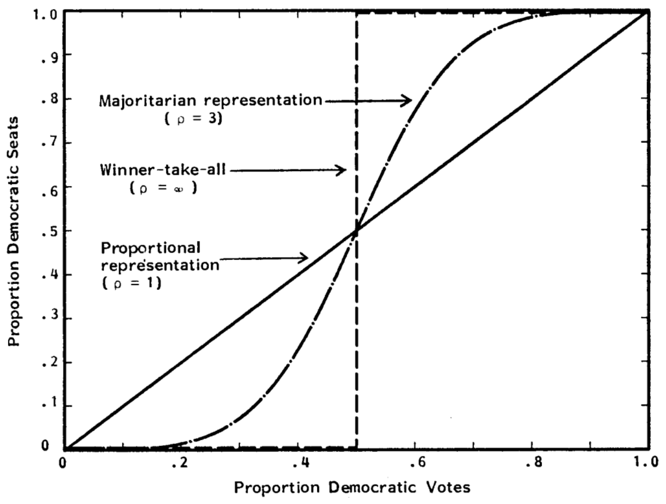

Figure 2: Different values of ρ result in different graph forms. When $\rho = 1$, the system is proportional, where each vote corresponds to a seat. When $\rho = \infty$, it represents a winner-takes-all system where any vote share above 50% results in all seats being won. The value $\rho = 3$ is known as majoritarian representation, where parties with majority votes are overrepresented, and minority votes are underrepresented. This most closely resembles how electoral systems function in reality.

Coefficient Value	Direction of Bias
$\beta >1$	Bias towards Republicans
$\beta = 1$	No Directional Bias
$\beta <1$	Bias towards Democrats

Table 1: How bias affects the model with respect to Figure 3.

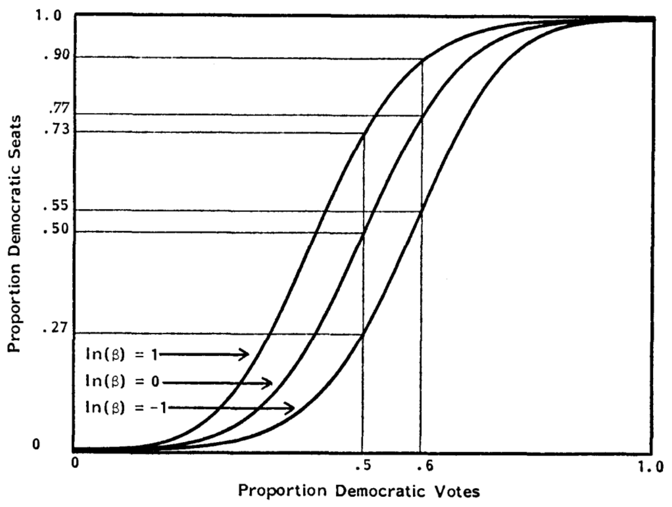

Figure 3: This graph shows different values of β with $\rho = 3$. Notice how bias towards Democrats results in them obtaining a higher seat share compared to a scenario with no bias.

Generally, β reflects asymmetry in the seats-votes relationship between two parties. Figure 3 and Table 1 illustrate how different values of β affect the model.

Now that we have a sufficient model to graph a state's electoral Seats/Votes relationship, we can use historical data to estimate the values of ρ and β and determine whether gerrymandering has occurred.

Gary King's analysis involved examining electoral data from 1950 to 1984 to estimate ρ and β for each state. This allowed him to graph the Seats/Votes relationship for each state and determine if gerrymandering was present.

Efficiency Gap

The **Efficiency Gap** is a method designed to measure fairness in electoral districting, particularly in identifying gerrymandering. It captures the difference in a party's respective **wasted** votes in an election.

A vote is considered wasted if:

	Party A	**Party B**
Total votes	550	450
Districts 1-3	70v / 20w	30v / 30w
Districts 4-8	54v / 4w	46v / 46w
Districts 9-10	35v / 35w	65v / 15w
Total w votes	150	350

Table 2: Party A vs Party B

- It is cast for a losing candidate.
- It is cast for a winning candidate but exceeds the necessary 50%.

The Efficiency Gap targets two gerrymandering strategies:

- *Cracking*: Diluting opposition votes across many districts.
- *Packing*: Concentrating opposition votes in a single district.

In any electoral system, inefficiencies are inevitable. For instance, in a two-party, single-member district system, 50% of votes are always wasted. The gerrymandering party doesn't need to eliminate wasted votes entirely; they only need fewer wasted votes than their opponent.

The Efficiency Gap is defined as the difference between the parties' respective wasted votes, divided by the total number of votes cast.

At its core, the Efficiency Gap summarizes all the 'cracking' and 'packing' decisions in an election into a single number:

$$\omega = \frac{W_A - W_B}{\sum v_i}$$

Where:

$$W_A = \sum \text{Surplus}_A + \sum \text{Lost}_A$$

$$W_B = \sum \text{Surplus}_B + \sum \text{Lost}_B$$

Example:

Consider two parties—Party A and Party B—across 10 uniform districts with 100 votes each.

The difference in wasted votes is 200. Therefore, the Efficiency Gap is $200/1000 = 0.2 = 20\%$.

This indicates that Party A wins 20% more seats than if the voting had equal weightage.

For scenarios with only two parties and districts of approximately equal size, the Efficiency Gap can also be calculated using:

$$\omega = \text{Seat Margin} - 2 \times \text{Vote Margin}$$

To effectively utilize the efficiency gap, a specific threshold is set, beyond which the result is considered unconstitutional. Typically, $|\omega| = 0.08$ is chosen as this threshold.

However, this method has a significant weakness: it can falsely flag perfectly symmetrical systems as gerrymandered. For instance, a vote/seat share of $0.65, 0.65$, where 65% of the votes result in 65% of the seats, would be incorrectly identified as gerrymandering against the other party.

$$\omega = 0.15 - 2 \times 0.15 = -0.15$$
$$|\omega| = 0.15$$

This happens because the efficiency gap suggests that the party with 35% of the votes is receiving a seat bonus greater than the winning party. According to the efficiency gap (EG) calculation, for a party to receive 35% of the seats, it should ideally receive 42.5% of the votes, as shown below:

$$\omega = 0.15 - 2 \times 0.075 = 0$$

Thus, while the efficiency gap is a useful measure for identifying bias, it is sometimes less reliable than other methods.

Markov- Chain Monte Carlo

Our goal is to use MCMC (Markov Chain Monte Carlo) to determine whether a state is engaging in gerrymandering. MCMC is named such because it uses Markov chains, where each state depends on the previous one.

What makes redistricting challenging is that each state has its own requirements and criteria that a plan must meet, such as ensuring each district has nearly equal population and maintaining certain shapes for districts. These complexities make it difficult to determine when changes to a redistricting plan constitute gerrymandering.

Let's consider a problem where a state has m geographical units (e.g., census blocks) that must be divided into n contiguous districts. We can formulate this as a redistricting graph cut problem, where an adjacency graph is partitioned into connected subgraphs.

Let $G = (V, E)$ represent the adjacency graph, where:

- $V = \{(1), (2), \ldots, (m)\}$ is the set of nodes.
- E is the set of edges connecting neighboring nodes, i.e., if two nodes $i, j \in V$ are contiguous, then the edge $(i, j) \in E$.

We partition the set of nodes V into n blocks, $\mathbf{v} = \{V_1, V_2, \ldots, V_n\}$, where $V_k \cap V_l = \emptyset$ and $\bigcup_{k=1}^{n} V_k = V$. Such a partition $\mathbf{v}$ generates an adjacency graph $G_{\mathbf{v}} = (V, E_{\mathbf{v}})$, where $E_{\mathbf{v}} \subset E$.

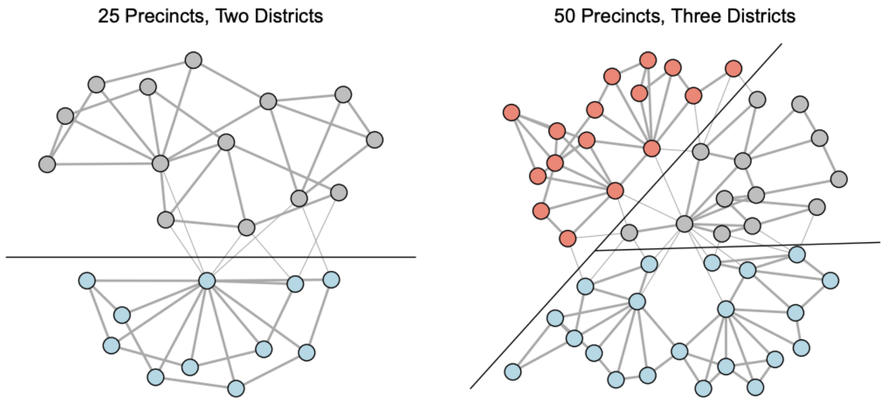

Figure 4: Redistricting as an Adjacency Graph Problem. The dark edges connect nodes within the same partition, while the light grey edges represent cut edges between different partitions. Different districts are represented in different colors.

4.2 The Basic Algorithm

To initialize the algorithm, we first obtain a valid partition $\boldsymbol{v}_0 = \{V_{1,0}, V_{2,0}, \ldots, V_{n,0}\}$ *and then repeat the following steps at each iteration* t*:*

Step 1 - "Turn on" edges:
Starting from the partition $\boldsymbol{v}_{t-1} = \{V_{1,t-1}, V_{2,t-1}, \ldots, V_{n,t-1}\}$*, obtain the adjacency graph* $G_{\boldsymbol{v}_{t-1}} = (V, E_{\boldsymbol{v}_{t-1}})$*. For every edge* $e \in E_{\boldsymbol{v}_{t-1}}$*, add* e *to* $E^*_{\boldsymbol{v}_{t-1}}$ *with probability* q*, creating* $E^*_{\boldsymbol{v}_{t-1}} \subset E_{\boldsymbol{v}_{t-1}}$*.*

Step 2 - Gather connected edges on boundaries:
Identify all components within $E^*_{\boldsymbol{v}_{t-1}}$ *that are connected to each other and adjacent to another block in the partition* $\boldsymbol{v}_{t-1}$*. Let* C *denote this set. This means that for all* $C_l \in C$*, there exists* $k \in \{1, 2, \ldots, n\}$ *such that* $C_l \cap V_{k,t-1}$ *and* $(i, j) \in E$ *for some* $i \in C_l$ *and* $j \in V_{k,t-1}$*.*

Step 3 - Select non-adjacent connected components:
Randomly select r *non-adjacent connected components from* C*, forming the set* C^**. Each component is selected with equal probability.*

Step 4 - Propose swaps:
Initialize the candidate partition $\boldsymbol{v}^*_t = \{V^*_{1,t}, V^*_{2,t}, \ldots, V^*_{n,t}\}$ *by setting* $V^*_{k,t} = V_{k,t-1}$*. For each component* $C^*_l \in C^*$*, find the block* $V_{k,t-1} \in \boldsymbol{v}_{t-1}$ *that contains* C^*_l*. Let* $A(C^*_l, \boldsymbol{v}_{t-1})$ *denote the set of blocks adjacent to* C^*_l*, excluding the block containing* C^*_l*. With probability* $1/|A(C^*_l)|$*, propose to assign* C^*_l *to an adjacent block. Update the blocks as follows:* $V^*_{k',t} = V^*_{k',t-1} \cup C^*_l$ *and* $V^*_{k,t} = V^*_{k,t} \setminus C^*_l$*. If* $V^*_{k,t} = \emptyset$*, return to Step 3. Note that after each swap,* $\boldsymbol{v}^*_t$ *remains a valid partition.*

Step 5 - Accept or Reject the proposal:
Using the Metropolis-Hastings algorithm, we set

$$\mathbf{v}_t = \begin{cases} \mathbf{v}_t^*, & \text{with probability } \alpha(\mathbf{v}_{t-1} \to \mathbf{v}_t^*), \\ \mathbf{v}_{t-1}, & \text{with probability } 1 - \alpha(\mathbf{v}_{t-1} \to \mathbf{v}_t^*). \end{cases}$$

The acceptance probability is given by

$$\alpha(\mathbf{v}_{t-1} \to \mathbf{v}_t^*) = \min\left(1, \frac{(1-q)^{|B(C^*, E_{\mathbf{v}_t^*})|}}{(1-q)^{|B(C^*, E_{\mathbf{v}_{t-1}})|}}\right).$$

where $B(C^*, E_{\boldsymbol{v}_t})$ is the set of edges in $E_{\boldsymbol{v}}$ that need to be cut to form connected components C^*.

4.3 Constraints

In practice, additional constraints are often introduced when running such algorithms, such as maintaining equal population between districts and ensuring geographical compactness. Here, we will focus on the population constraint and discuss how to efficiently incorporate it into our algorithm.

Assume p_i represents the population size in node i. A rough estimate of the population in each district, or the population parity, is given by

$$\bar{p} = \frac{\sum_{i=1}^{m} p_i}{n}.$$

The population equality constraint can be expressed as:

$$P_v = \max_{1 \le k \le n} \left| \frac{\sum_{i \in V_k} p_i}{\bar{p}} - 1 \right| \le \delta.$$

This constraint iterates through each district, summing the population of its nodes, and checks the deviation from the population parity $\bar{p}$. The maximum deviation across all districts is then compared to a threshold δ, such as $\delta = 0.03$, meaning that the population of any district cannot deviate from parity by more than 3%.

Instead of applying a strict constraint that could reject many candidate partitions and skew the final stationary distribution, we can adjust our Markov chain's stationary distribution to favor plans that meet the population constraint. We target the following distribution:

$$f_\beta(\mathbf{v}_t) = \frac{1}{z(\beta)} \exp\left(-\beta \sum_{V_l \in \mathbf{v}_t} \left| \frac{\sum_{i \in V_l} p_i}{\bar{p}} - 1 \right| \right),$$

where $z(\beta)$ is the normalizing constant and β is the inverse temperature. The negative exponent causes the distribution to peak when the populations of the districts are close to parity.

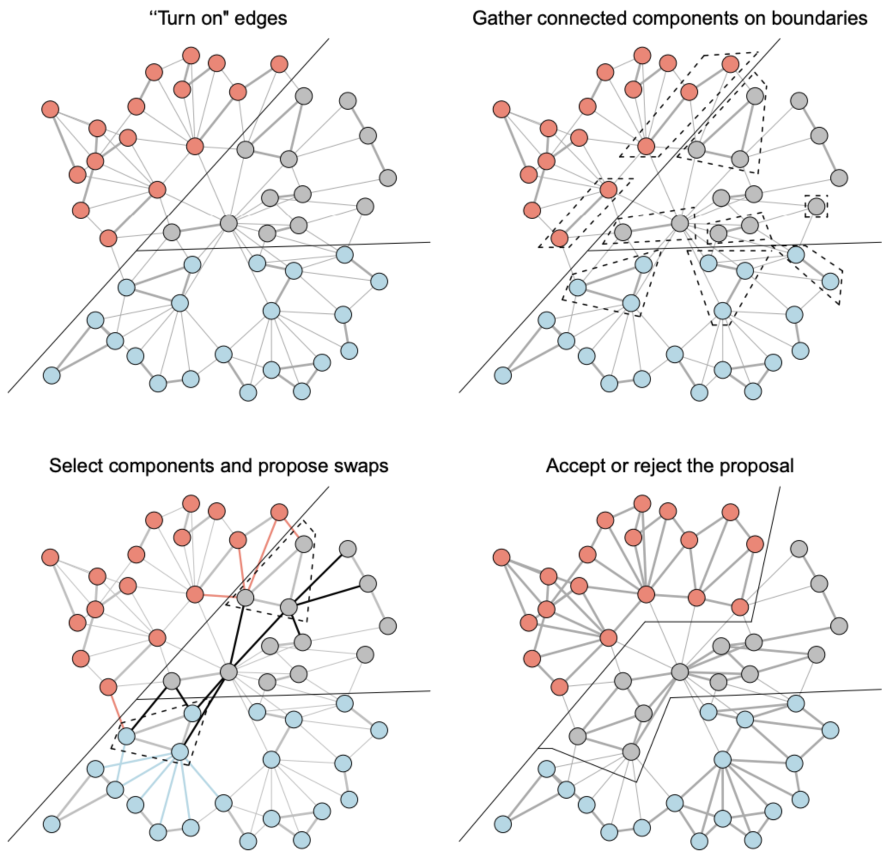
“Turn on" edges
Gather connected components on boundaries
Select components and propose swaps
Accept or reject the proposal

Figure 5: The Redistricting Algorithm. The plots describe the algorithm using the 50 Precinct, 3 district adjacency graph from Figure 4. The top-left graph represents edges being ”turned on” (light-grey) with independent probability. The top-right graph shows how edges adjacent to another precinct are found and selected. The bottom-left graph showcases the selection of a few adjacent edges, with one surrounding precinct chosen randomly to propose a swap. Finally, the bottom-right graph illustrates the final step, where the swaps are proposed and either accepted or rejected using the Metropolis-Hastings algorithm.

To incorporate this into our algorithm, we modify the acceptance probability as follows:

$$\alpha(\mathbf{v}_{t-1} \to \mathbf{v}_t^*) = \min\left(1, \frac{(1-q)^{|B(C^*, E_{\mathbf{v}_t^*})|} g_\beta(\mathbf{v}_t^*)}{(1-q)^{|B(C^*, E_{\mathbf{v}_{t-1}})|} g_\beta(\mathbf{v}_{t-1})}\right),$$

where $g_\beta(\mathbf{v}_t)$ is the unnormalized form of $f_\beta(\mathbf{v}_t)$, i.e., $g_\beta(\mathbf{v}_t) \propto f_\beta(\mathbf{v}_t)$.

4.4 Application in Code

To write code that demonstrates this algorithm, we will make several adaptations to apply it on a smaller scale. The original algorithm is designed for hundreds, if not thousands, of nodes, so the following changes will be made:

1. The code will run on a smaller set of nodes and districts, with the number of nodes ranging from 15-25 and districts from 3-7, randomly assigned.
2. Instead of selecting node edges and connecting adjacent components, we will focus on switching a single node, given the smaller node count.
3. The acceptance probability will be adjusted accordingly, but it will still follow the same logic and principles as the original.
4. A population constraint will be enforced, while population is evenly distributed across nodes, preventing any district from dominating all nodes.
5. Each node will be allocated a number of seats and assigned an initial political party, either Party A or Party B, while ensuring an equal number of nodes support each party.
6. In each district, if Party A holds a majority, all the district's seats are assigned to Party A; similarly for Party B. This ensures both parties have an equal chance before the redistricting algorithm begins.
7. To illustrate the seat distribution between the parties, after every switch, the number of seats held by Party A and Party B is calculated and added to a histogram, which will be displayed after all the changes.

These modifications retain the core logic of the MCMC gerrymandering algorithm but simplify it, making it feasible for a smaller scale and easier to demonstrate.

The R code implementing the algorithm is available in the attached GitHub repository.

References

[1] Fifield, Benjamin, et al. *A New Automated Redistricting Simulator Using Markov Chain Monte Carlo*, 20 July 2014.

[2] King, Gary, and Robert X. Browning. “Democratic Representation and Partisan Bias in Congressional Elections.” *American Political Science Review*, vol. 81, no. 4, Dec. 1987, pp. 1251–1273, doi:10.2307/1962588.

[3] McGhee, Eric. “Measuring Partisan Bias in Single Member District Electoral Systems.” *SSRN Electronic Journal*, 2013, doi:10.2139/ssrn.2195785.

[4] Austin, David. “Decoding, Gerrymanders, and Markov Chains.” *Feature Column*, 13 Nov. 2023, `https://mathvoices.ams.org/featurecolumn/2022/11/01/decoding-gerrymanders-and-markov-chains/`. Accessed Aug. 2024.

[5] Duchin, Moon. “Gerrymandering Metrics: How to Measure? What’s the Baseline?” *arXiv.Org*, 6 Jan. 2018, doi.org/10.48550/arXiv.1801.02064. Accessed Aug. 2024.

[6] Fifield, Benjamin, Michael Higgins, Kosuke Imai, et al. “Automated Redistricting Simulation Using Markov Chain Monte Carlo.” *Journal of Computational and Graphical Statistics*, vol. 29, no. 4, 7 May 2020, pp. 715–728, doi:10.1080/10618600.2020.1739532.