Analyzing Gerrymandering with Markov Chain Monte Carlo

Nick Verdoni, RJ Reilly, Masen Bachleda, Geige Zollicoffer

December 20, 2019

1 Introduction to MCMC

1.1 General Picture

In Nebraska, suppose it is possible to generate every single potential districting plan that could possibly exist, not only on the county level, but even down to the precinct level. Then, one could compare the current districting plan in question to every single potential districting plan and see how it stacks up against the others. This operation would be very fair as it relies not on human discrepancies and judgement but rather on statistical analysis. If the plan under question awards more seats to one particular party than the vast majority of all of the other plans that exist, this would raise a red flag that this district plan is very unfair and potential gerrymandering exists.

Unfortunately, for states such as Nebraska, it is not possible to generate every districting plan that exists down to the precinct level. Even with the best computers available, there are simply too many potential districting plans for the computers to generate in an even remotely timely fashion. However, it *is* possible to create a valid sampling measure to accurately represent the overall distribution of every districting plan that exists; this process is called Markov Chain Monte Carlo (MCMC). Utilizing the Markov Chain Monte Carlo process, we are able to generate an authentic and representative sampling of the entire distribution and, after the results have been obtained, we can effectively compare any district plan to it to test the plan's fairness.

1.2 MCMC Theory

To understand Markov Chains, one must first understand a finite state machine (FSM). An FSM is a type of program that goes from one "state" to another, with each destination on a finite list of possible states. An example of this would be a washing machine, which goes from the "idle" state to the "fill" state, to the "wash" state, to the "spin" state, and then finally back to the "idle" state. A more complicated FSM may be a program that's running traffic lights. It may

have a state for one road having a green light or another state for the crossing road having a green light while being protected by a turn arrow; however, there are conditions on when it goes from one state to the next.

A Markov Chain is operates similarly; however, instead of moving deterministically from state to state based on predetermined conditions, the Markov Chain moves randomly while also utilizing probabilities. For example, while an FSM for a traffic light may state *"If you're in the 'North/South green' state with the northbound turn lane occupied and no traffic on the southbound road, transition to 'North green and protected left turn'"*, a Markov chain diagram would simply declare *"If you're in the 'North/South green' state, then there's a 37.5 percent chance that the next state will be 'North green and protected left turn'."*

When one is developing the model for a Markov Chain, one enumerates the states (of which may exist a finite or infinite number) and then figures out what state will follow from any given state as well as with what probability [1]. For example, State A may have a 70 percent chance of leading to state B and a 30 percent chance of leading back to state A [1].

The most important characteristic of a Markov Chain model is that, at any point in the chain, the next step is independent of the previous step before it (as well as all other steps before it). This ensures that eventually, at some point in time t, the chain will be so far along that the current state at any point in time after t is sufficiently uncorrelated and unaffected by the starting state. Because of this, the sampling will be truly representative of the actual distibution [2].

2 Our Markov Chain Code

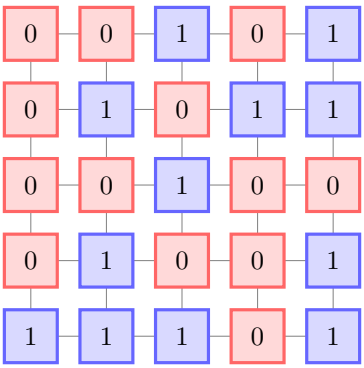
Over the course of the semester, our group constructed a Markov Chain Monte Carlo code using Python that can be used to analyze the voting districts of an M X N grid graph that represents a US state. (Before being able to directly apply this to the state of Nebraska, we first had to tackle some sample problems.) Below, the theory behind the code we created will be explained. At the end of the paper, the raw code can also be found in the Code Appendix.

2.1 Terminology

Voting Distribution

The smallest voting unit, a precinct, is represented on the graph with a node. In an M x N grid graph, there will be (M)(N) precincts that create the voting plane. In our simulation, each precinct votes either red (zero) or blue (one), and these votes do not change during the simulation. The way that each district votes is defined as the voting distribution.

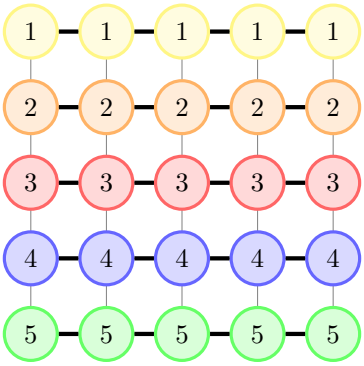
Here is an example of a voting distribution for a 5 x 5 graph.



Districting Plan

Precincts are grouped together to create a voting district. On the graph, a district is constructed by grouping nodes (precincts) of the same district together with edges. These edges connecting nodes must exist horizontally or vertically. In this simulation, each district must be contiguous, and every district must be comprised of an equal number of precincts. In an M x N voting plane graph, there will be M districts with N precincts in each district. The unique grouping of nodes (precincts) to form a series of districts is defined as a districting plan.

Here is an example of a districting plan for a 5 x 5 graph. In this 5 x 5 districting plan, there will be 5 districts with 5 precincts (nodes) in each district. These districts will be labeled 1, 2, 3, 4, or 5.

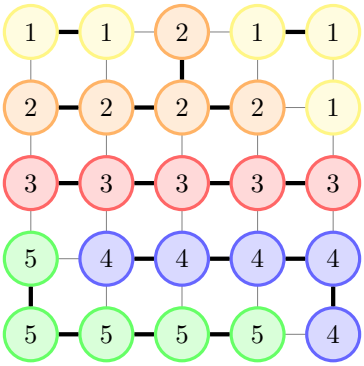


Contiguity

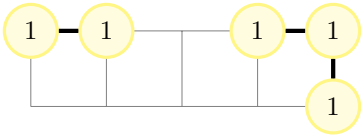
For a districting plan to be valid, there are two necessary conditions: every district must have the same number of votes and every district must be contiguous. For a district to be contiguous, its subgraph must be connected. By definition, for a subgraph of a district to be connected, it must contain a walk that connects every node within the district.

Consider the (invalid) districting plan below. Districts 2, 3, 4, and 5 are contiguous as there exists a singular walk connecting every node in the respective

district. However, District 1 is not contiguous as a singular walk connecting all nodes in the district does not exist. Because there exists a discontiguous district in the districting plan, the districting plan is invalid.



Additionally, analyzing the subgraph of District 1 illustrates the district's discontiguity as the subgraph of District 1 is disconnected.



Labeling Nodes

To make referring to nodes on the voting plane graph uniform, each node is assigned a Cartesian coordinate. The node in the bottom row and farthest left column is assigned the origin point (0,0). Working from the origin, to right horizontally is the positive X axis and upwards vertically is the positive y axis.

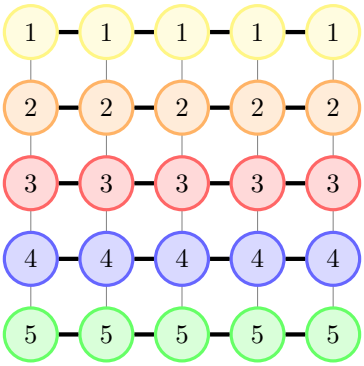
2.2 Code Operations

Initial Districting Plan

To begin the Markov Chain, an initial valid districting plan is needed for the program to intake. In general, the code intakes a starting M x N matrix with M districts of N votes each. The initial plan needs to be valid; thus, each of the M districts must have exactly N votes and all districts within the plan must be contiguous.

The theory behind the code will be demonstrated with a 5 x 5 districting plan. There are a multitude of 5 x 5 valid districting plans; any valid districting can be used as the starting plan. For example, one possible starting districting plan is the following generic districting plan:

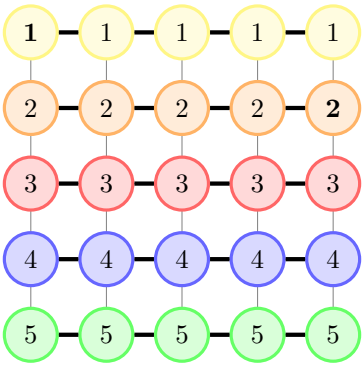
$$G_0 = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 & 4 \\ 5 & 5 & 5 & 5 & 5 \end{pmatrix}$$



Nodes Swaps

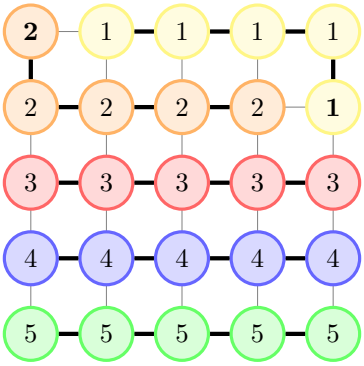
After selecting a valid starting districting plan, the program selects two random nodes to swap in attempt to create a new districting plan; this is the fundamental concept behind the Markov Chain. If the swap if valid and a legitimate districting plan is constructed, the code will accept this districting plan and proceed to the next phase of the algorithm. If the swap results in a faulty districting plan with one or more discontiguous districts, the code will discard this invalid plan and run the swap again.

Consider the graph G_0. *An example of a valid node swap is the bolded District 1 node at* (0,4) *and the bolded District 2 node at* (4,3).

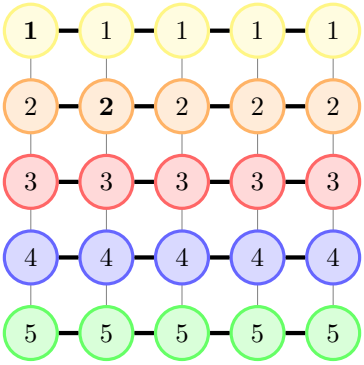


Completing the swap, the node in the first column and first row becomes labeled as "District 2" and the node from the fifth column and second row becomes labeled "District 1". In the code, each node is assigned a Cartesian coordinate. When a swap happens, the node's physical Cartesian coordinate does not change;

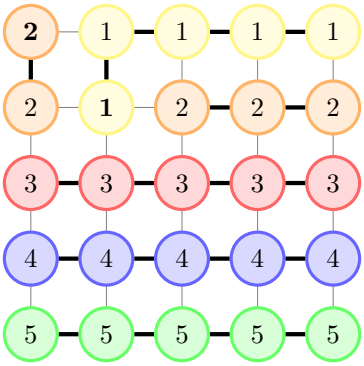
rather, the node's "attribute" changes instead. Thus, in this example, the node in the first column and first row now has attribute "District=1" while the other swapped node has "District=2". Because all five districts are still contiguous, our new districting plan is valid and will be accepted. The new graph will be labeled G_1.



As noted above, not all swaps result in a new valid districting plan; some swaps will lead to districts that are not contiguous. Let's analyze the districting plan that would occur if the bolded District 1 node at (0,4) *is swapped with the bolded Distict 2 node at* (1,3).



If these two nodes were swapped, the resulting districting plan would be invalid. Because a singular walk that connects all nodes in District 2 does not exist, District 2 is by definition discontiguous and the districting plan is invalid. This can also be visualized in the illustrated below. Thus, as is the case with all invalid swaps, the code will reject this swap and attempt to generate a different valid swap instead.



Note: The swapping of two nodes from the same district is considered an invalid swap and will be discarded.

Node Swap Selection List

The method of randomly selecting two nodes to swap in attempt to create a new valid districting plan is rather inefficient in and of itself because the number of valid swaps that exist pale in comparison to the large number of invalid swaps that exist. Furthermore, considering for each iteration the code will keep attempting node swaps until a valid districting plan is constructed and that the chain requires hundreds of thousands of iterations to run, the code - without any parameters to assist in narrowing down the potential node swaps to only valid ones - would take unrealistically long to run. Thus, a running list of swappable nodes is built into the code to improve efficiency.

The premise for this node swap selection list is as follows. After the initial districting plan is randomly generated, the code first identifies all potential valid swaps and compiles these pairs of nodes into a list. For the first iteration of the chain, the code will check every possible swap that exists for the given starting districting plan graph. Thus, in the 5 x 5 matrix case, the code compiles and analyzes all 300 possible swaps, selects the valid swaps, and compiles the valid swaps into a list.

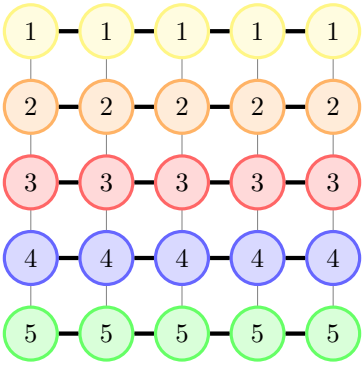
After compiling the list of node pairs that result in valid swaps, the code then makes a single random selection of one node pair and completes the swap. The new districting plan is then analyzed and its results are recorded. (This will be discussed in further detail later.) This is the end of the first iteration of the code.

Beginning the second iteration, this new districting plan will now be analyzed for potential nodes to swap to create the next districting plan. However, the code doesn't need to generate the valid node swap list from scratch. Instead, the code recalls the valid node swap list from the previous iteration and makes the following edits. If the node pair was not directly impacted by the previous swap, the pair remains in the list. If the node pair was effected by the previous swap, it is removed from the list. Then, the code analyzes the few remaining

new potential node swaps and adds the valid ones to the list.

The pairs of nodes that were not directly impacted by the previous node swap remain in the list and did not have to be regenerated. Thus, there is a heavy up front time cost during the first iteration of the code. However, because the chain doesn't have to recompile the entire list with each iteration and instead can modify the list as necessary, time efficiency is improved over the long run.

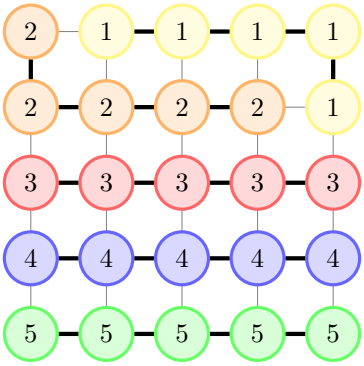
Consider the graph G_0. For the first iteration of the chain, the code would generate the following list of potential valid node swaps:



L_0 = {(0,4),(4,3); (0,3),(4,4); (0,3),(4,2); (0,2),(4,3); (0,2),(4,1); (0,1),(4,2); (0,1),(4,0); (0,0),(4,1)}

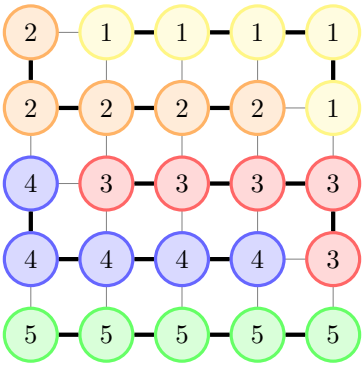
From this list, the code would randomly select a pair of nodes to swap. Let's select the node pair (0,4) *and* (4,3) *to swap; this new graph is the graph G_1.*

After the new graph G_1 is analyzed and the results recorded, the next iteration begins. Instead of compiling the node swap list for graph G_1 from scratch, the code will retain from the list L_1 the valid node pairs unaffected by the previous swap. (The retained nodes pairs are bolded in the list below.) All new potential node swap pairs are constructed, and if they result in a valid districting plan, are added to the list L_1. (The new node pairs are underlined in the list below.)



$L_1 = \{$**(0,4),(4,3)**; (1,4),(3,3); **(0,2),(4,1)**; **(0,1),(4,2)**; **(0,1),(4,0)**; **(0,0),(4,1)**$\}$

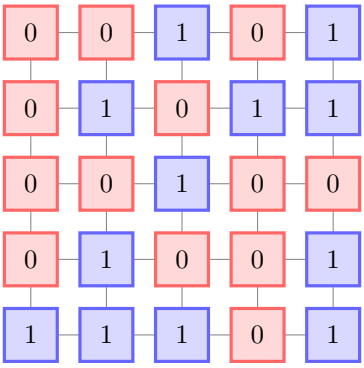
The code will now randomly select a pair of nodes to swap from the list L_1*. Let's select the node pair* (0,2) *and* (4,1) *to swap. This new graph will be label* G_2*.*

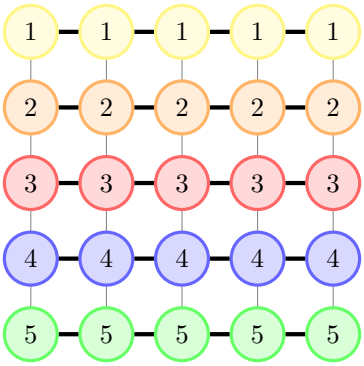


Scoring The Districting Plan

After a valid districting plan is generated, the code then scores the districting plan by totaling the number of seats won by each respective party. In essence, the districting plan is overlaid on the voting distribution and each district is totalled. The results are then recorded in a database, and this database is updated with every iteration.

Consider the sample voting distribution below and the provided districting plan.





District	Votes Blue	Votes Red	Winner
1	2	3	Red
2	3	2	Blue
3	1	4	Red
4	2	3	Red
5	4	1	Blue

In this districting plan, the Red Party won 3 seats and the Blue Party won 2 seats. For reference, Red won 13 of the 25 precinct votes and was awarded 3 of the 5 seats. In contrast, Blue won 12 of the 25 votes and was awarded 2 of the 5 seats. This information is stored in the database before the next iteration begins.

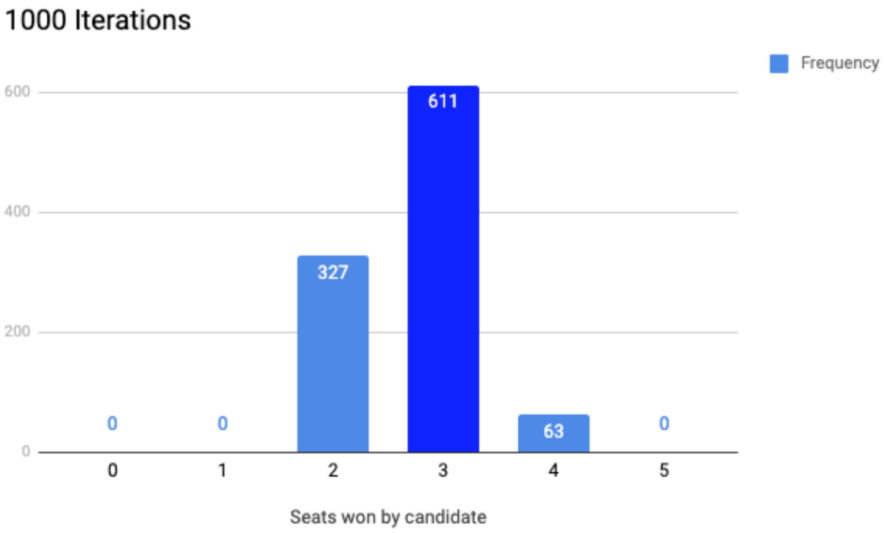
2.3 Code Process

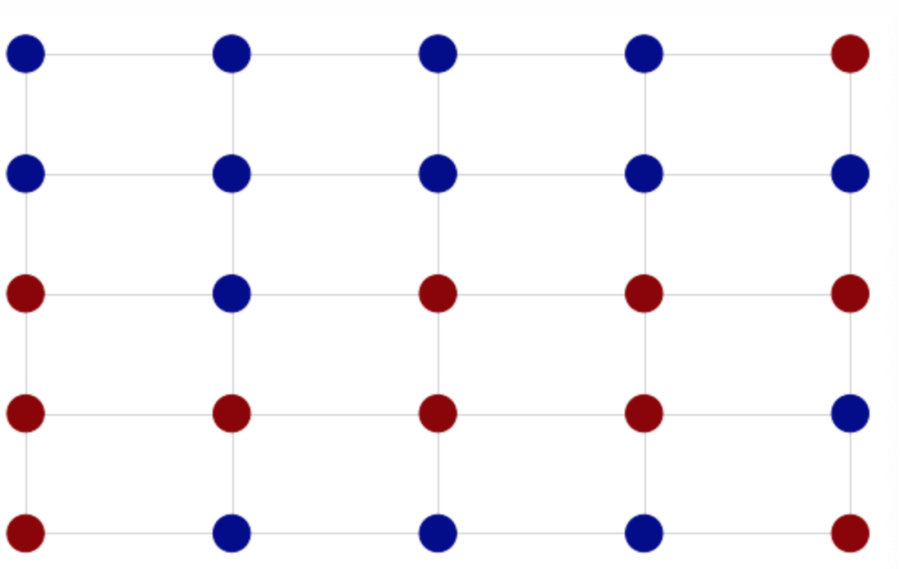
The previous subsections outlined the code's terminology and all of the operations that take place in the Markov Chain. Now it is time to put it all together. The premise of the chain is as follows. An initial districting plan is generated. The code then identifies the potential valid node swaps and compiles the running list. A random pair of nodes is selected, and the two nodes are swapped. This new valid districting plan is scored and the results are compiled in the database. This is the end of the first iteration of the chain. To begin the second iteration, the districting plan constructed in the previous iteration undergoes a node swap to generate a new districting plan. Before choosing the pair of nodes to swap, the code identifies and updates the running node swap list. Then, a pair of nodes are selected and this new districting plan is also scored in the same database. This is the end of the second iteration. This process continues until the desired number of iterations has been achieved. After the desired number of iterations has been accomplished, the results are compiled into a histogram for analysis.

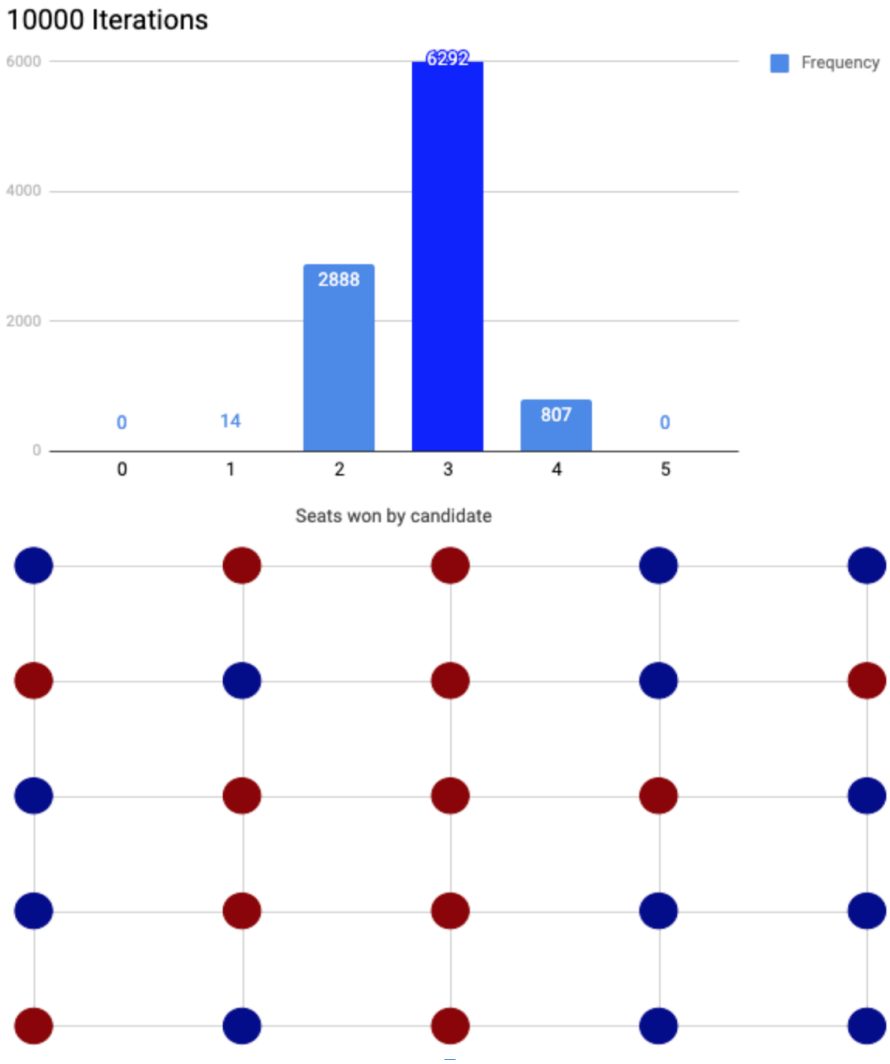
3 Authentic Simulation Results

The scope of this project was to use a sampling method, MCMC, to analyze if a districting plan is fair (or gerrymandered) by comparing the plan to a representative sampling of all possible districting plans that exist. To visualize the results, it is best to use the visual aid of a histogram. This histogram is created from the viewpoint of one particular party. (For example, in the results below, the histograms will be constructed from the viewpoint of the blue party and the number of seats they would win.) The horizontal axis of the histogram displays the number of seats won by the party and the vertical axis displays the number of districting plan iterations that resulted in the corresponding number of seats won. Typically, the results on the histogram create a bell curve or modified bell curve. When comparing the existing districting plan to the MCMC sampling, if the number of seats won by the party is an outlier (as evidence by it being on a tail of the curve), this is a major red flag and could potentially be evidence of gerrymandering.

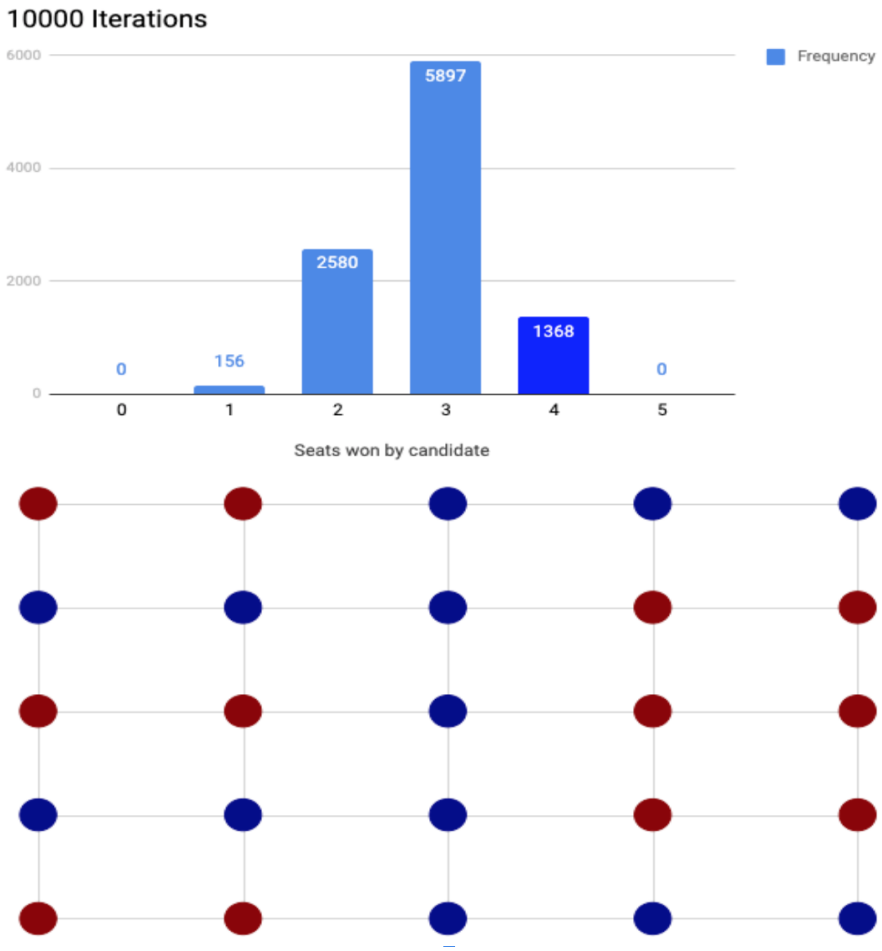
Below are some real results that we have generated from a simulation using our code. The districting plan that is being compared to the MCMC sampling is the 5x5 matrix with horizontal straight line districts (that has been used throughout this paper). On the histograms below, the highlighted bar represents where the original horizontal districting plan lies with respect to the number of seats won by blue. (In each of the four simulations below, we chose to change the voting distribution from the previous simulation to construct an even wider scope of samples. The voting distribution does not change at all during a single simulation).


1000 Iterations
Frequency
600
400
200
0
611
327
0
0
63
0
0
1
2
3
4
5
Seats won by candidate

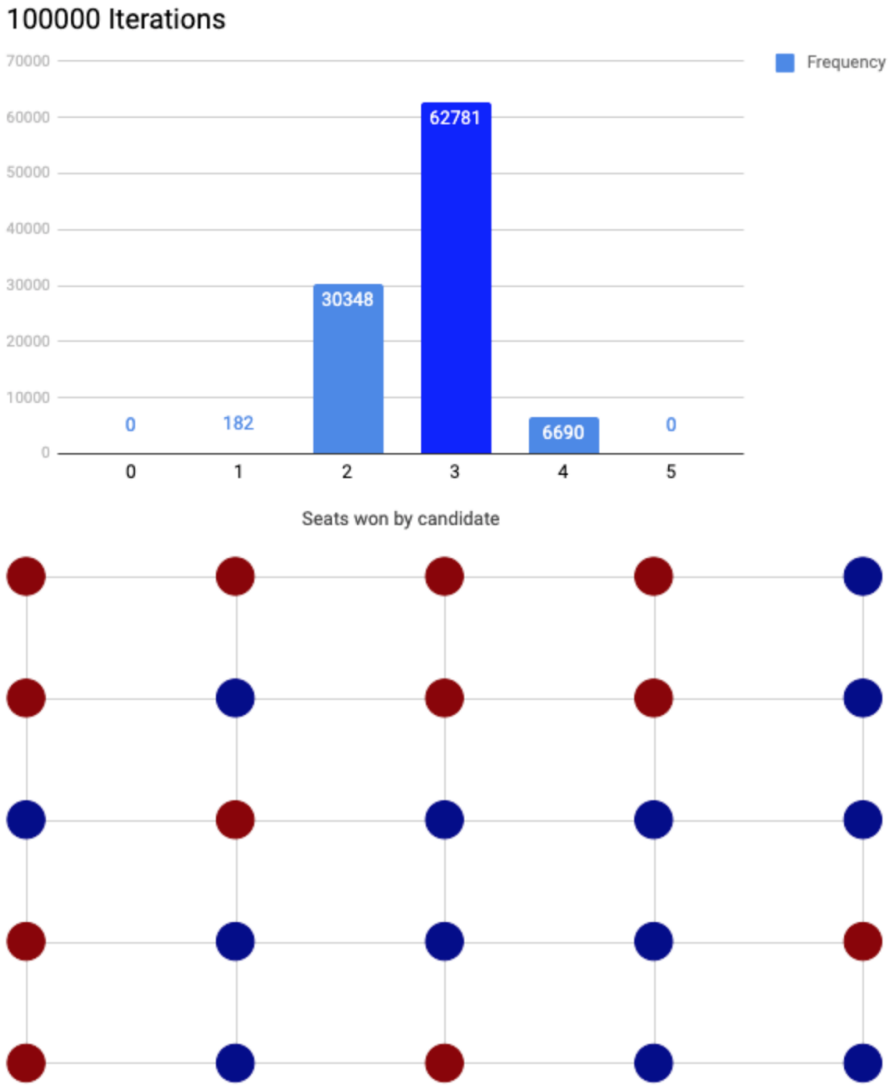




In this simulation that features 10000 iterations, Blue has 13 precinct votes and Red has 12 precinct votes. Blue wins 3/5 seats which seems fair. Thus, there does not appear to be evidence of gerrymandering.



In this simulation that also features 10000 iterations, Blue has 13 precinct votes and Red has 12 precinct votes. However, Blue wins 4/5 seats in this districting plan. Because this is an outlier, this could be potential evidence of gerrymandering.



In this simulation that features 100000 iterations, Blue has 13 precinct votes and Red has 12 precinct votes. Blue wins 3/5 seats in this districting plan. The plan does not appear to be gerrymandered.

4 Applying MCMC in Nebraska

Ultimately, our final goal was to apply our code analyze the districting plans in the state of Nebraska. The biggest challenge with this is the lack of voting data available for Nebraska, particularly on the precinct level, to properly simulate

the districting plans. If the data was to become available, spacial graphing programs such as QGIS are available to assist in converting the voting information into a graphical data set. However, even if one was able to obtain the proper data to construct a Nebraska voting graph, the other potential problem would be the run time needed for the simulation. Simply put, the larger the graph is, the more iterations are needed to run before one obtains a truly representative and valid sampling. Additionally, the larger the matrix is, the longer it takes to run each iteration and, in Nebraska, there are thousands of precincts that would compose the graph. Thus, because of the large size of the potential Nebraska graph and the need for more iterations, this may lead to some problems with run time (if it is even possible to run). For example, in the previous histogram with 100,000 iterations, the simulation took about an hour to run. To put this in perspective, the matrix was only a 5x5 matrix with 100,000 iterations. The matrix representing Nebraska would be thousands of times bigger, and we would have to run millions of iterations for the sampling to be valid. The framework is all there with our code, so it's not impossible. With the proper CPU's and with enough data, applying MCMC to analyze Nebraska's districting plans could definitely be accomplished.

References

[1] Victor Powell and Lewis Lehe. *Markov Chains Explained Visually.* `http://setosa.io/ev/markov-chains/`

[2] Sayantini Deb. *Explore Markov Chains With Examples — Markov Chains With Python.*

5 Code Appendix

5.1 Definitions

```
# -*- coding: utf-8 -*-
"""
Created on Thu Dec 19 12:49:24 2019

@author: GeighZ
"""

import numpy as np
import random as rd
import matplotlib.pyplot as plt
import networkx as nx
import math
import copy
import sys
```

```
from itertools import count
from IPython import display

"""
Created by Geigh Zollicoffer, Nick Verdoni, Masen Bachelda, Roderick Riley
Math 435 MCMC project

Simulates valid districting plans over a voter distribution
in order to test the fairness of a initial plan.
"""

def colorful_vertex_plot(g, pos, attr, node_size = 75, cmap = plt.cm.jet,
                         plot_title = ''):
    fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(10, 6))
    ax.set_title(plot_title)
    ax.axis('Off')
    groups = set(nx.get_node_attributes(g, attr).values())
    mapping = dict(zip(sorted(groups),count()))
    nodes = g.nodes()
    colors = [mapping[g.node[n][attr]] for n in nodes]
    ec = nx.draw_networkx_edges(g, pos, alpha=0.2)
    nc = nx.draw_networkx_nodes(g, pos, nodelist=nodes, node_color=colors,
                                with_labels=False, node_size = node_size,
                                cmap = cmap)

def round_sf(number, significant):
    return round(number, significant - len(str(number)))

def is_valid_swap(x, y,G):
    x_dist = G.nodes[x]['district']
    y_dist = G.nodes[y]['district']
    Dx = G.subgraph([node for node in G.nodes if G.nodes[node]['district'] ==
                     x_dist and not node == x])
    Dy = G.subgraph([node for node in G.nodes if G.nodes[node]['district'] ==
                     y_dist and not node == y])
    for comp in nx.connected_components(Dx):
        if set(G.neighbors(y)).intersection(set(comp))== set():
            return False
    for comp in nx.connected_components(Dy):
        if set(G.neighbors(x)).intersection(set(comp))== set():
            return False
    return True


def updateSwaps(G,x,y):
```

```
    nodes_to_check = [x]+[y]+list(G.neighbors(x))+list(G.neighbors(y))
    for node in G.neighbors(x):
        for nbr in G.neighbors(node):
            nodes_to_check.append(nbr)
    for node in G.neighbors(y):
        for nbr in G.neighbors(node):
            nodes_to_check.append(nbr)
    nodes_to_check = list(set(nodes_to_check))
    #Adding to Swaps for the corresponding node
    for z in nodes_to_check:
        G.nodes[z]['swaps'] = []
        for node in nodes:
            if G.nodes[node]['district'] == G.nodes[z]['district']:
                pass
            else:
                if is_valid_swap(z, node,G):
                    G.nodes[z]['swaps'].append(node)
                    if z not in G.nodes[node]['swaps']:
                        G.nodes[node]['swaps'].append(z)
                else:
                    if z in G.nodes[node]['swaps']:
                        G.nodes[node]['swaps'].remove(z)



class gerrymanderDataTable:

    def __init__(self,amountOfDist,distSize,graph):
        #per
        self.winThresh = math.floor(distSize/2.0)+1;
        #Current wins within a district
        self.districtWinsPool = dict.fromkeys((range(0,amountOfDist)));
        #Current wins for each party throughout the graph.
        self.partyWins = dict.fromkeys((range(0,amountOfDist)));
        #Amount of districts won for a party
        self.seatWins = [0] * (amountOfDist+1)
        self.setWinsPool(graph);

    def setWinsPool(self,graph):

        '''
        Set every key to 0 meaning blue is currently winning 0 seats
        in each district.
        '''
        for district in self.districtWinsPool:
```

```
            self.districtWinsPool[district] = [0]
        '''
        This is all time party wins within a certain district
        First item is blue wins, second is white wins,
        second is not currently being used since we
        are only displaying data of the blue party.
        '''
        for district in self.partyWins:

            self.partyWins[district] = [0,0]

        #This is all time wins for a [district], 0 means no seats.
        for district in self.seatWins:
            self.seatWins[district] = 0



        #Just setting up what the win total currently looks like:
        for node in graph.nodes:
            dist = graph.nodes[node]['district'];
            if(graph.nodes[node]['party']==0):
                self.districtWinsPool[dist][0] +=1

        blueSeats = 0;
        for district in self.districtWinsPool:
            if(self.districtWinsPool[district][0]>=self.winThresh):
                self.partyWins[district][0] +=1
                blueSeats+=1
        self.seatWins[blueSeats] = 1

    def getSeatWins(self):
        return self.seatWins

    def updateTable(self,swapX,swapY,g):

        '''
        Gaining data between two precincts that were swapped
        between district X and district Y, hence the name
        swap X and swap Y
        '''

        xDist = g.nodes[swapX]['district']
        xParty = g.nodes[swapX]['party']
        yDist = g.nodes[swapY]['district']
        yParty = g.nodes[swapY]['party']
```

```
        '''
        Update table on if 0/blue party won a precinct,
        If the party won: decrease previous district total number
        of blue/0 party wins by 1 and increase new district's
        total number of blue/0 precinct wins by 1.
        '''

        if(xParty == 0):
            self.districtWinsPool[xDist][0] +=1
            self.districtWinsPool[yDist][0] -=1

        if(yParty == 0):
            self.districtWinsPool[yDist][0] +=1
            self.districtWinsPool[xDist][0] +=-1

        blueSeats = 0;

        for district in self.districtWinsPool:
            if(self.districtWinsPool[district][0]>=self.winThresh):
                blueSeats+=1

        self.seatWins[blueSeats] +=1
```

5.2 Setup

```
# -*- coding: utf-8 -*-
"""
Created on Thu Dec 19 12:55:23 2019

@author: GeighZ
"""

'''
These are settings that a user can use to manipulate the size
of the districting plans.
'''
distSize = 6;
amountOfDist = 6;

#Our Toy graph coresponding to toy parameters.
g = nx.grid_graph(dim=[amountOfDist,distSize])
validSwaps = [];

#setting up the graph with certain attributes
#party = 0  means party = blue
```

```
blueCounter = 0
'''
for i in range(0,amountOfDist):
    for j in range (0,distSize):
        g.nodes[(j,i)]['district']=i
        g.nodes[(j,i)]['swaps']=[]

        #Random Party Assignment.
        #randint = rd.randint(0,1)
        if i == 0 or i == amountOfDist-1 or j == 0 or j== amountOfDist-1:
            coin = np.random.binomial(1,.4)
            g.nodes[(j,i)]['party'] = coin;
        #g.nodes[(j,i)]['party']= randint
        #if(randint == 0):
            if coin == 0:
                blueCounter +=1
        else:
            g.nodes[(j,i)]['party'] = 0;
            blueCounter+=1
votePer = (blueCounter/(distSize*amountOfDist))*100
print("blue holds %",votePer,"Percent of the vote")
'''
for i in range(0,amountOfDist):
    for j in range (0,distSize):
        g.nodes[(j,i)]['district']=i
        g.nodes[(j,i)]['swaps']=[]

        #Random Party Assignment.
        #randint = rd.randint(0,1)
        if i == 0 or i == amountOfDist-1 or j == 0 or j== amountOfDist-1:
            coin = np.random.binomial(1,.4)
            g.nodes[(j,i)]['party'] = coin;
        #g.nodes[(j,i)]['party']= randint
        #if(randint == 0):
            if coin == 0:
                blueCounter +=1
        else:
            g.nodes[(j,i)]['party'] = 0;
            blueCounter+=1
votePer = (blueCounter/(distSize*amountOfDist))*100
print("blue holds %",votePer,"Percent of the vote")
'''
Gathers and appends all possible swaps for each precinct
'''

pos = dict((node, node) for node in g.nodes)
```

```
nodes = list(g.nodes)
for i in range(len(nodes)):
    for j in range(i, len(nodes)):
        if g.nodes[nodes[i]]['district']==g.nodes[nodes[j]]['district']:
            pass
        else:
            if is_valid_swap(nodes[i],nodes[j],g):
                g.nodes[nodes[i]]['swaps'].append(nodes[j])
                g.nodes[nodes[j]]['swaps'].append(nodes[i])

#Halts program until user hits enter.
input("Press Enter to continue...")
nodes = list(g.nodes)

#Our object that will keep track of statistics.
gdt = gerrymanderDataTable(amountOfDist,distSize,g);

#Running the update on valid swaps and making the swap.
swaps = []
for node in nodes:
    for swap in g.nodes[node]['swaps']:
        if{node,swap} not in swaps:
            swaps.append({node,swap})

someArr = gdt.getSeatWins()
x = list(range(0,amountOfDist+1,1))
x_pos = [i for i, _ in enumerate(x)]
plt.bar(x_pos, someArr, color='green')
plt.xlabel("Seats Won")
plt.ylabel("Frequency")
plt.title("Seats won by blue")
plt.xticks(x_pos, x)
```

5.3 Driver

```
# -*- coding: utf-8 -*-
"""
Created on Thu Dec 19 12:55:40 2019

@author: GeighZ
"""

'''
This section of the data will make the random swaps
while keeping statistics of districting plans.
This is the main part of the code with an
```

```
optional animated bar plot.
'''

#iterations: a user can manipulate n for any amount of swaps.
n = 5000
t = 0
p= n*.01
enumX = list(range(0,amountOfDist+1,1))
while t<n:
    #Optional percentage tracker/animated bar plot
    if t%p==0:
        per = (t/n)*(100)
        sys.stdout.write("\r%f%%" %per)
        sys.stdout.flush()
        if t%10 ==0:
            plt.clf()
            seatWins = gdt.getSeatWins()
            x_pos = [i for i, _ in enumerate(enumX)]
            plt.bar(x_pos, seatWins, color='blue')
            display.display(plt.gcf())
            display.clear_output(wait=True)
    #Part of the code that handles swaps:
    proposal = list(rd.choice(swaps))
    x = proposal[0]
    y = proposal[1]
    proposal_G = copy.deepcopy(g)
    x_dist = proposal_G.nodes[x]['district']
    y_dist = proposal_G.nodes[y]['district']
    proposal_G.nodes[x]['district'] = y_dist
    proposal_G.nodes[y]['district'] = x_dist
    error = updateSwaps(proposal_G,x,y)
    new_swaps = []
    for node in nodes:
        for swap in proposal_G.nodes[node]['swaps']:
            if {node,swap} not in new_swaps:
                new_swaps.append({node,swap})
    p = min([len(swaps)/len(new_swaps),1])
    coin = np.random.binomial(1,p)
    #Gives a probability on staying with same districting plan,
    #once valid swaps are calculated.
    if coin == 1:
        g = proposal_G
        swaps = new_swaps
        t+=1
        error = gdt.updateTable(x,y,g)
    else:
```

```
        t+=1
        error = gdt.updateTable(x,x,g)


'''
Final output of histogram, votes distribution and exact count
of votes for blue/0 in every district.
'''

seatWins = gdt.getSeatWins()
x = list(range(0,amountOfDist+1,1))
x_pos = [i for i, _ in enumerate(x)]
plt.bar(x_pos, seatWins, color='blue')
plt.xlabel("Seats Won")
plt.ylabel("Frequency")
plt.title("Seats won by blue")
plt.xticks(x_pos, x)
print("\n")
for i in range(0,amountOfDist+1):
    print("Blue won ",i," seats ",seatWins[i],"times.")
colorful_vertex_plot(g, pos, 'party', node_size=500)

'############################EOF############################################################'
```