How to build districting ensembles: A guide to GerryChain

Daryl DeFord

May 31, 2019

Contents

1 Introduction **2**
- 1.1 Disclaimers . . . 2
- 1.2 Outline . . . 2
- 1.3 Discrete Redistricting . . . 2
- 1.4 Ensemble Analysis . . . 3
- 1.5 Other Resources . . . 3

2 GerryChain Fundamentals **3**
- 2.1 Installing GerryChain . . . 4
- 2.2 MCMC Implementation . . . 4
- 2.3 The Partition Class . . . 5
- 2.4 The Markov Chain Iterator . . . 5

3 Building a Dual Graph **6**
- 3.1 Geospatial Data . . . 6
- 3.2 Starting Plans . . . 7

4 Updaters **7**
- 4.1 Elections . . . 8
- 4.2 Splitting Rules . . . 9
- 4.3 Partitions Revisited . . . 9

5 Binary Constraints **10**
- 5.1 Contiguity . . . 10

6 Proposal Distributions **10**
- 6.1 Boundary Flips . . . 11
- 6.2 ReCombination . . . 11

7 Acceptance Functions **11**
- 7.1 Annealing . . . 12

8 Running the Chain **12**

9 Auxilliary Material **15**

1 Introduction

This guide provides an introduction to using the GerryChain software, developed by the Voting Rights Data Institute (VRDI) and the Metric Geometry and Gerrymandering Group (MGGG), for generating ensembles of districting plans. Computational approaches to political redistricting have become increasingly important as access to new resources and techniques have revolutionized the study of districting plans. In particular, Markov chain sampling and Markov chain Monte Carlo (MCMC) methods are being applied both in court cases and legislative reform efforts and this open–source implementations offers a flexible toolkit for performing these analyses.

This guide contains many code examples that can be run in any python interpreter once the software is installed. Full running versions of the templates described in the text are available for download here.

1.1 Disclaimers

One of the guiding principles behind the GerryChain project was to develop a flexible and modular implementation of MCMC on graph partitions. More specifically, we do not want the software to force you to perform specific types of analysis. Thus, while we provide a basic framework, it is likely that you will need to implement some of your own tools in order to carry out the analyses that are most interesting to you. Indeed, our thinking on the most important considerations has evolved over time and will undoubtedly continue to evolve as we learn more about the problem. With this in mind, this guide will demonstrate how to construct your own functions to supplement the basic packages.

The software is also under constant development and while we will try to keep this material updated it is possible that some of the commands will change over time. You can follow the development of the project here. The overall logical structure of the implementation will remain fixed and so the high level concepts described below will still allow you to understand how to interact with GerryChain even if the terminology changes.

Finally, this guide is focused quite closely on actually running GerryChain and assumes that you have some basic experience with Markov chains, graphs, and the redistricting problem. The documents linked in Section 1.5 provide some additional background for these topics and material for gaining a broader understanding of the underlying principles.

1.2 Outline

The remainder of this section provides some brief background on the redistricting problem, the uses of ensemble analysis, and other writing on this topic that might be useful. Section 2 walks you through the installation process and describes the two fundamental python classes that make GerryChain tick: the Partition class and the Markov Chain iterator. That section also describes the all of the procedures that are carried out at each step of the Markov chain. The next five sections explore each of these steps in detail with code snippets and examples of writing new functions. Finally, Section 8 shows how to actually run the chain and process the information at each step while Section 9 provides some details on storing, analyzing and plotting the outputs from the chain runs.

1.3 Discrete Redistricting

For our purposes, we will view redistricting as a discrete problem, where the region to be divided is partitioned into units that are individually assigned to districts. For actual states these units are likely to be either census units like blocks or tracts or state defined units like precincts. Given a partition of the geography, we form a *dual graph* by associating a node to each unit and placing an edge between two units if they share a common boundary. This equates the districting problem with graph partitioning, a well–studied problem from both the theoretical and applied perspective.

Once we have our graph, we must decide which partitions are permissible. This is usually guided by the laws of the state under consideration and various federal guidelines. Sadly, these laws are rarely mathematically specific enough to have a unique interpretation, instead choices must be made in order to interpret the rules in such a way that we can measure their properties on a given graph partition. Common examples include the contiguity of districts, which we will enforce by requiring the induced subgraphs of the districts

be connected graphs and population balance, which we measure by comparing the sum of the populations of the nodes in each district to the ideal population of a district. Other constraints include various compactness scores, rules about spliting counties and municipalities, compliance with the Voting Rights Act, and preservation of communities of interest among others.

Once the laws have been operationalized we can set about to try and construct a partition that satisfies the constraints we have identified. There are many popular methods for optimizing the graph partitioning problem subject to some collection of constraints. However, for our purposes it is not sufficient to construct a single partition, instead we are interested in analyzing the statistical properties of large collections of partitions all defined on the same dual graph. Examples of this method are described in the next section.

1.4 Ensemble Analysis

One of the original uses of ensemble analysis for districting plans was to evaluate the impact of partisan intent of proposed or enacted plans. The basic idea of was to construct a large collection of plans, generated by a "neutral" computer that did not take in to account any partisan information, and compare the expected partisan performance of these plans to that of the human drawn plan under consideration. If the original plan is an extreme outlier under some collection of partisan imbalance metrics this provided evidence that the plan was drawn with partisan intent. This has played out in expert reports in court cases across country, all the way to the Supreme Court.

Although the main application of ensemble methods to this point has occurred in the adversarial setting of the courtroom, there are many other applications. Our group has been active in using ensembles not just to criticize enacted plans but to better understand the underlying political geography of individual states and evaluate the impacts of proposed legislation on the types of plans that are likely to be enacted. Examples of our work can be found here.

1.5 Other Resources

First of all, the official documentation for GerryChain is quite good and provides several examples of chain runs in addition to the documentation of the individual functions and components. As mentioned above, this guide is mostly focused on the technical implementation of GerryChain and the processes that are necessary to generate ensembles in practice. For background material on discrete Markov chains and MCMC my notes here are a good place to start along with the corresponding webpage and GitHub Repository for interactive tools exploring the ideas. Section 7 of that document also describes the application to redistricting analysis specifically and provides greater theoretical details than this guide.

I taught an IAP course at MIT this January and the course website has many additional resources for exploring aspects of this problem. It also includes the lecture slides from the course which provide a great deal of background information and several state specific examples of ensemble analysis. Some fun tools for playing around with districting on grids can be found at these links: (basic) and (advanced).

Additional information about the geospatial data processing necessary to generate cleaned shapefiles for GerryChain (and other applications) is available here, the documenation for the MGGG maup package is also quite helpful and examples of our cleaned data are available on GitHub. Information about open mathematical problems related to redistricting is available at this webpage.

2 GerryChain Fundamentals

This section describes the important features and classes of the GerryChain package at a high level as a guide for the remainder of the document. GerryChain lives on Github here and you can follow the development process by keeping an eye on recent commits and the issues tab. If you want to contribute to the project, we recommend that you read the contributor guidelines and fork the repo to your account as we operate on the pull request system.

2.1 Installing GerryChain

We currently recommend using Conda to install GerryChain. The main difficulty in getting a working version of GerryChain is that the necessary GIS dependencies are frequently updated, causing compatibility issues. If you already have geopandas, pyproj, fiona, and shapely you should be able to install from PyPi:

```
>>> pip install gerrychain
```

Assuming that you are starting from scratch, begin by installing MiniConda from this link. Choose the Python 3.7 version that matches your operating system and install from the binary that you download. After installing MiniConda, on Unix and Max systems you can open a regular command line on Windows you need to open a separate "Anaconda Prompt" from the start menu. Begin by creating an environment for gerrychain:

```
>>> conda create -n Gerry
>>> conda activate Gerry
```

You should see (Gerry) to the right of your cursor in the command line. Python environments are a way to manage package installation for specific applications without forcing them to be installed globally. You can read more about conda environments here. Now you can install the gerrychain package and all of its dependencies at once:

```
>>> conda update conda
>>> conda init
>>> conda install -c conda-forge gerrychain
```

Once this process has completed you can check the installation by entering the following into the command line:

```
>>> python
>>> import gerrychain
```

If this doesn't generate any error messages, congratulations! You are now ready to generate ensembles. If it does, condolences! Send me an email with your operating system, what you tried, and the text of the error message and we'll try to figure it out.

2.2 MCMC Implementation

This section describes the tasks that are carried out at each step of our implementation of MCMC on graph partitions. To see more details of the reasoning behind this formulation, see Section 7 of this pdf. Before constructing an ensemble, the geospatial data must be processed to generate the underlying dual graph. Then, an initial state must be constructed, either from the a proposed plan associated to the data or by growing a new "seed plan" that satisfies the constraints of the chain. Section 3 discusses these concepts in more detail. Finally, you have to decide what it is that you want to measure on each of the generated partitions. These measurements are recorded in *updater functions* that takes as input the current partition and output a value of collection of values such as partisan imbalance scores or the population of each district. Section 4 describes how to construct these functions.

Next we describe the actual MCMC process implemented in GerryChain. Given the current state in our chain to advance to the next state we perform the following steps:

1. Generate a new proposed state according to the *proposal function*
2. Check that the proposed state is valid by evaluating *validity checks* on the the proposal
    - If the state passes all checks proceed to 3
    - Else return to 1
3. Apply an *acceptance function* to the proposed state

- If accepted the proposal becomes the next state in the chain
- Else the current state becomes the next state in the chain

4. Apply the updaters to measure the statistics of the new plan.

The three key function classes that are explored in detail in the Sections below are the *proposal function* which takes as input the current partition and outputs a dictionary that maps nodes to their new proposed assignments, *validity checks* which take as input the proposed partition and returns a binary True/False, and the *acceptance function* which takes as input the current state and proposed state and returns a binary True/False.

2.3 The Partition Class

The truly fundamental object of GerryChain is the Partition class, which represent a single partition of the given graph, along with the relevant statistics that we are trying to measure. A Partition consists of three objects: the underlying dual graph of the state which contains annotations like population, area, or partisan information, an assignment dictionary that maps the nodes of the graph to districts, and a collection of updaters which record the statics of interest for the partition. More details on each of these components are provided in the sections below.

The syntax for creating a Partition is below, we haven't built or imported any of these other objects yet but soon will:

```
>>> my_initial_partition = Partition(graph = Dual_Graph,
        assignment = Initial_Assignment, Updaters = My_Updaters)
```

Throughout the remainder of the document we will explore how to extract useful information from this object. As a brief example, to get the collection of land areas of each district in the current plan you can type:

```
>>>my_initial_partition["areas"]
```

For each updater you construct, the values it returns are accessed in a similar fashion.

2.4 The Markov Chain Iterator

To generate the appropriate partitions, we construct a Markov chain object, which is an iterator that returns the Partition object generated at each step of the Markov chain. The Markov chain takes as input an Partition object as defined above, a proposal function, list of validity checks, an acceptance function, and the number of steps to run the chain.

The syntax for the MarkovChain looks like:

```
>>>    my_chain = MarkovChain(
    proposal=my_proposal,
    constraints=my_list_of_checks,
    accept=my_acceptance_function,
    initial_state=my_initial_partition,
    total_steps=1000
        )
```

To interact with the states of the chain we iterate over the partitions that are generated:

```
    for part in my_chain:
        do_things(part)
```

Once we have developed some more details we will look at this in more detail.

3 Building a Dual Graph

The first thing we need in order to build partition of a graph is the graph itself. Python provides support for graph objects using the NetworkX package and our graph objects have a similar set of properties and features. In addition to the adjacency information recorded in the graph we also want to make use of several node and edge attributes like area and perimeter that are associated to the graph. We view these attributes as fixed properties of the graph and support indexing based on the graph object itself, so the population of node 10 in the graph can be accessed as:

```
graph.nodes[10]["population"]
```

The most common use case for constructing a graph is directly from a shapefile that already has the relevant demographic and partisan information as described in the next section. However, you can also construct arbitrary non–spatial graphs directly. The following snippet constructs a 20×20 grid graph where each edge has length 1, each node has population 1, and each node either votes for the pink party (with 40% probability) or the purple party. The last line builds an assignment dictionary that partitions the grid into two horizontal columns.

```
>>> graph=nx.grid_graph([20,20])
>>> for e in graph.edges():
>>>         graph[e[0]][e[1]]["shared_perim"]=1
>>> for n in graph.nodes():
>>>         graph.node[n]["population"]=1
>>>         if random.random()<.4:
>>>             graph.node[n]["pink"]=1
>>>             graph.node[n]["purple"]=0
>>>         else:
>>>             graph.node[n]["pink"]=0
>>>             graph.node[n]["purple"]=1
>>>         if 0 in n or 19 in n:
>>>             graph.node[n]["boundary_node"]=True
>>>             graph.node[n]["boundary_perim"]=1
>>>          else:
>>>             graph.node[n]["boundary_node"]=False
>>> assignment_dict = {x: int(x[0]/10)  for x in graph.nodes()}
```

3.1 Geospatial Data

The usual procedure for constructing a dual graph is to begin with a shapefile, which is a proprietary data format that contains both geographic and tabular information, merged together. We maintain a collection of cleaned shapefiles here and more details about their construction and structure are available here. Assuming that you have downloaded and extracted the Pennsylvania shapefile you can access the columns as a pandas dataframe with geopandas:

```
>>> import geopandas as gpd
>>> df = gpd.read_file("./PA_shapefile.shp")
```

The list of available data columns is available here. Notice that it includes population, demographcs, partisan information, and proposed plans, everything we need to get started building ensembles. GerryChain provides tools for extracting dual graphs along with the relevant data columns from shapefiles or dataframes directly, by computing the set of adjacent units from the geography, as well as the ability to construct them from .json files. The .json approach requires you to have computed the adjacencies previously but is much quicker for subsequent uses.

```
>>> from gerrychain import Graph
>>> import json
>>> import
```

```
>>> dual_graph = Graph.from_file("./PA_shapefile.shp")
>>> dual_graph2 = Graph.from_geodataframe(df)
>>> with open('./PA_graph.json', 'w') as outfile:
        json.dump(json_graph.adjacency_data(dual_graph), outfile)
>>> dual_graph3 = Graph.from_json("./PA_graph.json")
```

These graphs now have the adjacency structure of the precincts of the state as well as the relevant information for running our MCMC.

3.2 Starting Plans

In addition to the scalar values representing demographic and geogrpahic information of our graph nodes, we also need to have an initial assignment of nodes to districts to start the Markov chain. We provide tools to extract these directly from the graph or shapefile. For example, the Pennsylvania data that we loaded above has eight proposed plans pre-loaded and we can use the corresponding column as a starting point:

```
>>> assignment = "REMEDIAL_P"
```

While the cleaned data usually includes enacted plans as defined by the census, these plans may not always satisfy the constraints we wish to study. In this case we can construct our own assignment dictionary as with the grid graph example above. More sophisticated tools for developing initial seeds are in development. However, we do provide some methods that recursively apply contiguous bipartitioning algorithms to build individual districts to within some population tolerance:

```
>>> from new_seeds import recrsive_tree_part
>>> number_districts = 18
>>> population_column = "TOT_POP"
>>> tolerance = .01
>>> new_seed = recursive_tree_part(graph,number_districts,population_column,tolerance,1)
>>> df["new_seed"]=df[id_col].map(new_seed)
>>> print(df.groupby(["new_seed"])[population_column].agg('sum'))
```

This returns a new partitioning of Pennsylvania into 18 Congressional districts with the populations balanced to within 1% and prints the population values of each district.

4 Updaters

For our intended applications, it isn't sufficient to generate a large collection of plans, we also want to make measurements of the properties of those plans at every step in the Markov chain. In GerryChain, many of these measurements, particularly those that are necessary for informing the proposals, validity checks, and acceptance functions, are handled with updaters. At the most basic level, an updater is a function that takes as input a Partition and returns a collection of values that are then accessible from the Partition itself.

An updater consists of a string that tells the Partition what to call the value and a function that is evaluated on the Partition object at every step of the chain. The GeographicPartition class maintains some updaters automatically, such as the area, perimeter and GerryChain provides some others such as the set of dual graph edges that lie between districts, known as cut edges. These can be accessed from the partition object by their names. For example,

```
>>> partition["cut_edges"]
```

returns the set of boundary edges. As a simple example, we will write an updater that also tracks the number of cut edges. We start by writing a function that takes as input a partition and returns this number and then add it to our set of updaters:

```
>>> def number_of_cut_edges(partition):
>>>     return len(partition["cut_edges"])
>>> boundary_length_updater = {"number_cut_edges": number_of_cut_edges}
>>> my_updaters.update(boundary_length_updater)
```

Here is a simple example that tracks the number of steps in the Markov chain, which is useful for performing annealing:

```
>>> def step_num(partition):
>>>     parent = partition.parent
>>>     if not parent:
>>>         return 0
>>>     return parent["step_num"] + 1
>>> step_updater = {"Step": step_num}
```

Another illustrative example is the population of each district in the partition. We need access to this value in order to decide whether or not the given partition satisfies our population tolerance, so it must be formulated as an updater so our population validity check has access to the values. Many updaters take this form, as a sum of some column over all nodes that belong to a specific district, and so we provide a Tally function that automatically sums the given column name over all nodes in each district separately:

```
>>> population_updater = {"population": updaters.Tally("TOT_POP", alias="population")}
>>> my_updaters.update(population_updater)
```

Tallys return a dictionary whose keys are the labels of the districts and whose values are the corresponding sum.

Now if we construct a partition with these updaters we can extract the populations and length of boundary for the initial state:

```
>>> my_initial_partition = Partition(graph = dual_graph,
        assignment ="REMEDIAL_P", Updaters = my_updaters)
>>> my_initial_partition["number_cut_edges"]
>>> my_initial_partition["population"]
```

Later on, when we actually run the chain, it is these values that we will store and analyze across the entire ensemble.

4.1 Elections

As one of the most commonly studied features of these ensembles concerns the expected partisan performance of the districting plans we provide a separate class for handling election data. These special updaters take as input the name of the election, as well as the columns and names of the corresponding vote counts for two parties:

```
>>> election_2008_updater = Election(
    "2008 Senate",
    {"Democratic": "2008_D", "Republican": "2008_R"}
        )
>>> my_updaters.update(election_2008_updater)
```

You can attach as many election updaters as you like, for some states we have dozens of elections worth of data. Once the updater is added to the partition, it can be used to return some additional types of analysis of the election. In the background, the election updater computes two separate Tallys, one for each party and then computes the winner and percentages for each district:

```
>>> my_initial_partition["2008 Senate"].wins("Republican")
>>> my_initial_partition["2008 Senate"].percents("Democratic")
```

returns the number of districts won by the Republicans using that election and districting plan and the percentage of voters that voted for the Democratic party in each district, respectively. The partisan imbalance measures are formulated to accept these Election objects directly. For example:

```
>>> mean_median(my_initial_partition["2008 Senate"])
```

will return the Mean–Median value associated to the current districting plan and selected vote column.

4.2 Splitting Rules

Several states have rules that incentivize keeping counties or other municipal units together. Thus, it is sometimes useful to track the number of counties that are split by a the current districting plan. Although there is an updater implemented to compute these values (upaters.countysplits, which takes the name of the county column as an input) there is a more efficient way to perform this computation if you have access to the geodataframe with the same columns as your dual graph.

As above, we define a function that takes as input the partition:

```
>>> def num_splits(partition, df=df):
>>>     df["current"] = df.index.map(partition.assignment)
>>>     return sum(df.groupby('COUNTYFP10')['current'].nunique() >1)
>>>  my_updaters.update({"County Splits": num_splits})
```

Similar approaches can be used to simplify other computations that are based on the columns available to the dataframe. The partial function provided by functools can also be used to allow you to pass the name of the splits column to the function for additional flexibility.

4.3 Partitions Revisited

Now that we have dual graphs, initial plans and updaters we can build an actual partition object. Here is an example from some of our recent work in Alaska:

```
#Import Data
>>> df = gpd.read_file("./data/AK_precincts_ns/AK_precincts_ns/AK_precincts_ns.shp")
>>> df["nAMIN"] = df["TOTPOP"]-df["AMIN"]
#Build Updaters
>>> elections = [
    Election("GOV18x", {"Democratic": "GOV18D_x", "Republican": "GOV18R_x"}),
    Election("USH18x", {"Democratic": "USH18D_x", "Republican": "USH18R_x"}),
    Election("GOV18ns", {"Democratic": "GOV18D_NS", "Republican": "GOV18R_NS"}),
    Election("USH18ns", {"Democratic": "USH18D_NS", "Republican": "USH18R_NS"}),
    Election("Native_percent", {"Native":"AMIN", "nonNative":"nAMIN"})
        ]
>>> my_updaters = {"population": updaters.Tally("POPULATION", alias="population")}
>>> election_updaters = {election.name: election for election in elections}
>>> my_updaters.update(election_updaters)
#Construct Dual Graph
>>> dual_graph = Graph.from_file(
    "./data/AK_precincts_ns/AK_precincts_ns/AK_precincts_ns.shp"
        )
>>> dual_graph.join(df,columns= ["nAMIN"])
>>> idict={}
>>> for index, row in df.iterrows():
>>>  idict[int(row["ID"])] = index
#Attach Islands
>>> to_add = [(426,444), (437,438),(437,442),(411,420),
(411,414), (411,358),(411,407),(399,400),(399,349),(381,384),(240,210)]
>>> for i in range(len(to_add)):
>>>  dual_graph.add_edge(idict[to_add[i][0]],idict[to_add[i][1]])
>>>  dual_graph[idict[to_add[i][0]]][idict[to_add[i][1]]]["shared_perim"]=1
#Build Partition
>>> initial_partition = GeographicPartition(dual_graph, assignment="HDIST",
    updaters=my_updaters)
```

5 Binary Constraints

Once we have built our initial partition it is time to examine the components of the actual MCMC process. We begin with the binary constraints that define the state space. At each step of the chain, each proposal is evaluated to make sure that it is permissible. This is carried out by calling a sequence of functions called validity checks, that take as input the proposed partition and return True if it satisfies the constraint and False if it fails. These validity check functions are collected in a list that is provided to the Markov chain iterator as a single Validator object.

We provide a variety of helper functions for defining your own contraints. For numerical data, there are upper and lower bounds as well as a bound that forces the value to be within some percentage range of an ideal value, which is useful for population. For example, if we want the partitions to be balanced to within 5% of ideal and to have no more than 250 cut edges, we initialize:

```
>>> population_constraint = constraints.within_percent_of_ideal_population(
    initial_partition, .05)
>>> cut_edges_constraint = constraints.UpperBound(number_of_cut_edges, 250)
```

The bounds take the name of a function to evaluate on the partition and a value to compare the result to. There are also self-configuring bounds that use the initial partition to determine a bound based on the initial seed. For example, if we want the number of cut edges to be no worse than that of the initial partition we could use:

```
>>> cut_edges_constraint = SelfConfiguringUpperBound(number_of_cut_edges)
```

The validity checks do not necessarily have to be of this form. For example, we might instead never want two nodes to belong to the same district. We can enforce this by defining a function that returns False if they have the same assignment:

```
>>> def not_together(partition);
>>>     return partition.assignment["node1"] != partition.assignment["node2"]
```

and simply add this to the list of constraints for the Markov chain:

```
>>> constraints = [population_constraint, cut_edges_constraint, not_together]
```

Then, every time a new plan is proposed in the chain, each of these functions will be called and the proposal will only proceed if it returns True on all of them. A complete list of the included constraints can be found in the documentation.

5.1 Contiguity

One of the things that makes the generation of new plans difficult is requiring that the individual districts be contiguous. We provide a collection of different graph algorithms that check the contiguity of each district. The main versions are contiguous which uses pre-built NetworkX function, contiguousbf which uses a breadth–first search and singleflipcontiguous which is optimized for boundary flip proposals. We also provide a novanishingdistricts function for runs where the population balance is not constrained, so that there are the same number of districts at each step.

As an example, one set of recent experiments about the connection between self–avoiding walks on grids and the boundary flip proposal used the following as the constraints:

```
>>> constraints = [single_flip_contiguous, no_vanishing_districts]
```

6 Proposal Distributions

The heart of our MCMC methods are the proposal distributions that attempt to modify the current partitition. These functions take as input the current state and return a dictionary of (node, district) pairs of assignments to change for the new districting plan. Since they take as input the Partition object, they

have access to all of the updaters and can use these values to help select which assignments to change. Designing new proposal distributions that move efficiently through the space of partitions is an interesting and important research problem.

We will start by implementing a simple proposal that simply selects a random node and changes its assignment to a random district.

```
>>> def bad_proposal(partition):
>>>     node = random.choice(list(partition.graph.nodes()))
>>>     new_part = random.choice(list(partition.parts))
>>>     return partition.flip({node: new_part})
```

We can also change the assignment of many nodes at once. Here is a proposal that attempts to change the assignment of every node on the boundary of a single district:

```
>>> def mediocre_proposal(partition):
>>>     flip_label = random.choice(list(partition.parts))
>>>     flips = {}
>>>     for node in partition.graph.nodes():
>>>         if partition.assignment[node] == flip_label:
>>>             flips[node] = random.choice(list(
        partition.assignment[x] for x in partition.graph.neighbors(node)]))
>>>     return partition.flip(flips)
```

Neither of these is likely to return particularly useful results but they demonstrate the basic principles of proposal functions.

6.1 Boundary Flips

We provide several implementations of the boundary flip procedure. These all attempt to change the assignment of a single node that lies on the boundary of a district. The function proposerandomflip is the most efficient implementation but the resulting distribution is not quite reversible or uniform. A proposal that fixed reversibility is available as slowreversiblepropose, which as the name suggests is not quite as fast. Uniformity can be obtained by incorporating a waiting-weighting function that returns an empty dictionary (no flips) with some probability that depends on the number of adjacent permissible partitions.

6.2 ReCombination

We also provide an implementation of the ReCombination proposal described in our Virgina Report. Briefly, this proposal merges the nodes of two adjacent districts to form a subgraph and the bipartitions them in a contiguous fashion. The main implementation uses spanning trees to do the bipartitioning but there is also a spectral version implemented and several more on the way as described here.

This proposal requires a little additional setup as the ReCom step needs the ability to pass some additional information about the graph to its sub-functions:

```
>>> ideal_population = sum(initial_partition["population"].values())
>>> ideal_population = ideal_population / len(initial_partition)
>>> tree_recom = partial(recom,
                         pop_col="TOT_POP",
                         pop_target=ideal_population,
                         epsilon=0.05,
                         node_repeats=1
                              )
```

7 Acceptance Functions

Once a new proposal has passed all of the validity checks, it is passed to the acceptance function to determine whether the chain should move to the new state or remain in place. Like the validity checks, the acceptance

function takes as input the current partition and returns a True or False value. However, acceptance functions are usually comparing some feature of the current proposal to that of the previous state in the chain. We refer to the previous state as the parent partition and its properties can be accessed from partition.parent. For example, the boundary length at the previous step is:

```
>>> partition.parent["number_cut_edges"]
```

Note that the initial state does not have a parent, so we include a test at the beginning of each acceptance function to avoid comparing to this non–existent object. Here is a sample acceptance function always accepts a proposal that decreases the number of cut edges and only accepts a proposal that increases the number of cut edges 1% of the time:

```
>>> def edge_accept(partition):
>>>     bound = 1
>>>     if partition.parent is not None:
>>>         if partition.parent["number_cut_edges] > partition["number_cut_edges"]:
>>>             bound = .01
>>>     return random.random() < bound
```

7.1 Annealing

If we want to adjust the proposal as the chain progresses, such as in the setting of simulated annealing, we can use the stepnum updater described above to have the bound depend on the current step. Here is an annealing example from the experiments reported on in Section 7 here:

```
>>> def annealing_cut_accept(partition):
>>>     boundaries1  = {x[0] for x in partition["cut_edges"]}.union(
    {x[1] for x in partition["cut_edges"]})
>>>     boundaries2  = {x[0] for x in partition.parent["cut_edges"]}.union(
    {x[1] for x in partition.parent["cut_edges"]})
>>>     t = partition["step_num"]
>>>     if t <100000:
>>>         beta = 0
>>>     elif t<400000:
>>>         beta = (t-100000)/100000 #was 50000)/50000
>>>     else:
>>>         beta = 3
>>>     bound = 1
>>>     if partition.parent is not None:
>>>         bound = (base**(beta*(-len(partition["cut_edges"])+
    len(partition.parent["cut_edges"])))*(len(boundaries1)/len(boundaries2))
>>>     return random.random() < bound
```

8 Running the Chain

Now that we have all the components, we can actually build the Markov Chain iterator. Here is an example making use of the functions we defined above:

```
>>> my_chain = MarkovChain(
    proposal=tree_recom,
    constraints=[population_constraint, cut_edges_constraint, not_together]
    accept=edge_accept,
    initial_state=my_initial_partition,
    total_steps=100000
    )
```

Running this code will generate a Markov chain object that will generate states following the procedure described in Section 2. To access the individual states we have to actually iterate over the chain in a loop:

```
>>> for current_partition in my_chain:
>>>     do_something(current_partition)
```

Generally, we are interested in measuring some of the properties of each partition. For example, we might want to know how many seats the Democratic party won, the efficiency gap, what the length of the boundary was, and the sorted populations of the districts. We can record these values in lists:

```
>>> D_wins = []
>>> EGs = []
>>> boundary_length = []
>>> pops = []
>>> for current_partition in my_chain:
>>>     D_wins.append(current_partition["2008 Senate"].wins("Democratic")
>>>     EGs.append(efficiency_gap(current_partition["2008 Senate"]))
>>>     boundary_length.append(current_partition["number_cut_edges"])
>>>     pops.append(sorted(current_partition["population"].values()))
```

We can evaluate any function that takes a partition object on the states of the chain. For example, our initial partition did not include the county splits updater but we can still measure the number of splits for each step:

```
>>> for current_partition in my_chain:
>>>     print(num_splits(current_partition))
```

This allows us to interact with the data after the chain has finished. For long chain runs it is frequently convenient to write out the values to file occasionally in order to keep the memory requirements reasonable. Here is an example of this from some recent analysis on Virginia and you can see the full set of outputs on GitHub:

```
votes=[[] for x in range(num_elections)]
mms=[]
egs=[]
hmss=[]
split_vec = []

print("finished_initial_plot")
t=0
count=0
df=gp.read_file(plot_path)
df["START"]=df.index.map(assignment)

df.plot(column="START",cmap="tab20")

plt.savefig(newdir+"initial_plot.png")

plt.close()

for part in chain:


    pop_vec.append(bvap_vector(part,"population"))
    cut_vec.append(len(part["cut_edges"]))
    split_vec.append(num_splits(part))
    mms.append([])
```

```
egs.append([])
hmss.append([])

for elect in range(num_elections):
    votes[elect].append(bvap_vector(part,election_columns[elect][0]+"%"))
    mms[-1].append(mean_median(part, election_columns[elect][0]+"%"))
    egs[-1].append(efficiency_gap(part, col1=election_columns[elect][0],
     col2=election_columns[elect][1]))
    hmss[-1].append(how_many_seats_value(part, col1=election_columns[elect][0],
     col2=election_columns[elect][1]))

t+=1
if t%2000==0:
    print(t)
    with open(newdir+"mms"+str(t)+".csv",'w') as tf1:
        writer = csv.writer(tf1,lineterminator="\n")
        writer.writerows(mms)

    with open(newdir+"egs"+str(t)+".csv",'w') as tf1:
        writer = csv.writer(tf1,lineterminator="\n")
        writer.writerows(egs)

    with open(newdir+"hmss"+str(t)+".csv",'w') as tf1:
        writer = csv.writer(tf1,lineterminator="\n")
        writer.writerows(hmss)

    with open(newdir+"pop"+str(t)+".csv",'w') as tf1:
        writer = csv.writer(tf1,lineterminator="\n")
        writer.writerows(pop_vec)

    with open(newdir+"cuts"+str(t)+".csv",'w') as tf1:
        writer = csv.writer(tf1,lineterminator="\n")
        writer.writerows([cut_vec])

    with open(newdir+"splits"+str(t)+".csv",'w') as tf1:
        writer = csv.writer(tf1,lineterminator="\n")
        writer.writerows([split_vec])

    with open(newdir+"assignment"+str(t)+".json", 'w') as jf1:
        json.dump(part.assignment, jf1)


    for elect in range(num_elections):
        with open(newdir+election_names[elect]+"_"+str(t)+".csv",'w') as tf1:
            writer = csv.writer(tf1,lineterminator="\n")
            writer.writerows(votes[elect])

    df["plot"+str(t)]=df.index.map(part.assignment)
    df.plot(column="plot"+str(t),cmap="tab20")
    plt.savefig(newdir+"plot"+str(t)+".png")
    plt.close()

    votes=[[] for x in range(num_elections)]
    mms=[]
```

```
        egs=[]
        hmss=[]
        pop_vec=[]
        cut_vec=[]
        split_vec=[]
```

9 Auxilliary Material

Having generated some data from our chain we now want to visualize it or do some deeper analysis. GerryChain itself does not provide any plotting functionality - even more than not wanting to tell you how to set up your chain we don't want to tell you what to do with your data. That said, we do make a large number of visualizations of our own data. The following snippets from the Alaska project take as inputs lists of values like the example above and generate boxplots and histograms.

```
    partisan_w = [wins1,wins2,wins3,wins4]
    partisan_p = [percents1,percents2,percents3,percents4]
    p_types=["GOV18N", "GOV18A", "USH18N", "USH18A"]
    p_vecs=[GOV18x, GOV18ns, USH18x, USH18ns]
for y in range(4):
    plt.figure()
    plt.boxplot(np.array(partisan_p[y]),whis=[1,99],showfliers=False, patch_artist=True,
                    boxprops=dict(facecolor="None", color=c),
                    capprops=dict(color=c),
                    whiskerprops=dict(color=c),
                    flierprops=dict(color=c, markeredgecolor=c),
                    medianprops=dict(color=c),
                    )
    plt.plot(range(1,41),p_vecs[y],'o',color='red',label='Current Plan')
    plt.plot([.5,41],[.5,.5],color='green',label="50%")
    plt.xlabel("Sorted Districts")
    plt.ylabel("Dem %")
    plt.xticks([1,20,40],['1','20','40'])
    plt.legend()
    fig = plt.gcf()
    fig.set_size_inches((20,10), forward=False)
    fig.savefig("./Outputs/plots/Ensemble_Box_"+ types[z] + p_types[y] + ".png")
    plt.close()
print("Finished ",types[z]," Box plots")
for y in range(4):
    plt.figure()
    sns.distplot(partisan_w[y],kde=False,color='slateblue',bins=[x for x in range(10,25)],
                                                    hist_kws={"rwidth":1,"align":"left"})
    plt.axvline(x=sum([val>.5 for val in p_vecs[y]]),color='r',
        label="Current Plan",linewidth=5)
    plt.axvline(x=np.mean(partisan_w[y]),color='g',label="Matchings Mean",linewidth=5)
    plt.legend()
    print(p_types[y],"wins: ", np.mean(partisan_w[y]))
    plt.savefig("./Outputs/plots/Ensemble_Hist_"+ types[z] + p_types[y] + ".png")
plt.close()
```

Examples of the full plotting software that takes as input the values written to file are available on GitHub.