GerryChain Documentation

Metric Geometry and Gerrymandering Group

Apr 17, 2019

Contents

GerryChain is a library for using Markov Chain Monte Carlo methods to study the problem of political redistricting. Development of the library began during the 2018 Voting Rights Data Institute (VRDI).

The project is in active development in the mggg/GerryChain GitHub repository, where bug reports and feature requests, as well as contributions, are welcome.

CHAPTER 1

Installation

1.1 Using conda (recommended)

To install GerryChain from conda-forge using conda, run

```
conda install -c conda-forge gerrychain
```

For this command to work as intended, you will first need to activate the conda environment that you want to install GerryChain in. If the environment you want to activate is called `vrdi` (for example), then you can do this by running

```
conda activate vrdi
```

If this command causes problems, make sure conda is up-to-date by running

```
conda update conda
conda init
```

For more information on using conda to install packages and manage dependencies, see Getting started with conda.

1.2 Using `pip`

To install GerryChain from PyPI, run `pip install gerrychain` from the command line.

This approach often fails due to compatibility issues between our different Python GIS dependencies, like `geopandas`, `pyproj`, `fiona`, and `shapely`. For this reason, we recommend installing from conda-forge for most users.

CHAPTER 2

User Guide

These guides show you how to get started using GerryChain:

2.1 Getting started with GerryChain

This guide will show you how to start generating ensembles with GerryChain, using MGGG's Pennsylvania shapefile.

2.1.1 What you'll need

Before we can start running Markov chains, you'll need to:

- Install `gerrychain` from PyPI by running `pip install gerrychain` in a terminal.
- Download and unzip MGGG's shapefile of Pennsylvania's VTDs from GitHub.
- Open your preferred Python environment (e.g. JupyterLab, IPython, or a `.py` file in your favorite editor) in the directory containing the `PA_VTD.shp` file from the `.zip` that you downloaded and unzipped.

2.1.2 Creating the initial partition

In order to run a Markov chain, we need an adjacency *Graph* of our VTD geometries and `Partition` of our adjacency graph into districts. This Partition will be the initial state of our Markov chain.

```
from gerrychain import Graph, Partition, Election
from gerrychain.updaters import Tally, cut_edges

graph = Graph.from_file("./PA_VTD.shp")

election = Election("SEN12", {"Dem": "USS12D", "Rep": "USS12R"})

initial_partition = Partition(
```

(continues on next page)

(continued from previous page)

```
    graph,
    assignment="2011_PLA_1",
    updaters={
        "cut_edges": cut_edges,
        "population": Tally("TOT_POP", alias="population"),
        "SEN12": election
    }
)
```

Here's what's happening in this code block.

The *Graph.from_file()* classmethod creates a *Graph* of the precincts in our shapefile. By default, this method copies all of the data columns from the shapefile's attribute table to the `graph` object as node attributes. The contents of this particular shapefile's attribute table are summarized in the mggg-states/PA-shapefiles GitHub repo.

Next, we configure an `Election` object representing the 2012 Senate election, using the `USS12D` and `USS12R` vote total columns from our shapefile. The first argument is a name for the election (`"SEN12"`), and the second argument is a dictionary matching political parties to their vote total columns in our shapefile. This will let us compute hypothetical election results for each districting plan in the ensemble.

Finally, we create a `Partition` of the graph. This will be the starting point for our Markov chain. The `Partition` class takes three arguments:

graph A graph.

assignment An assignment of the nodes of the graph into parts of the partition. This can be either a dictionary mapping node IDs to part IDs, or the string key of a node attribute that holds each node's assignment. In this example we've written `assignment="2011_PLA_1"` to tell the `Partition` to assign nodes by their `"2011_PLA_1"` attribute that we copied from the shapefile. This attributes holds the assignments of precincts to congressional districts from the 2010 redistricting cycle.

updaters An optional dictionary of "updater" functions. Here we've provided an updater named `"population"` that computes the total population of each district in the partition, based on the `"TOT_POP"` node attribute from our shapefile, and a "SEN12" updater that will output the election results for the `election` that we set up. We've also provided a `cut_edges` updater. This returns all of the edges in the graph that cross from one part to another, and is used by `propose_random_flip` to find a random boundary node to flip.

With the `"population"` updater configured, we can see the total population in each of our congressional districts. In an interactive Python session, we can print out the populations like this:

```
>>> for district, pop in initial_partition["population"].items():
...     print("District {}: {}".format(district, pop))
District 3: 706653
District 10: 706992
District 9: 702500
District 5: 695917
District 15: 705549
District 6: 705782
District 11: 705115
District 8: 705689
District 4: 705669
District 18: 705847
District 12: 706232
District 17: 699133
District 7: 712463
District 16: 699557
District 14: 705526
```

(continues on next page)

(continued from previous page)

```
District 13: 705028
District 2: 705689
District 1: 705588
```

Notice that `partition["population"]` is a dictionary mapping the ID of each district to its total population (that's why we can call the `.items()` method on it). Most updaters output values in this dictionary format.

For more information on updaters, see the *gerrychain.updaters* documentation.

2.1.3 Running a chain

Now that we have our initial partition, we can configure and run a *Markov chain*. Let's configure a short Markov chain to make sure everything works properly.

```
from gerrychain import MarkovChain
from gerrychain.constraints import single_flip_contiguous
from gerrychain.proposals import propose_random_flip
from gerrychain.accept import always_accept

chain = MarkovChain(
    proposal=propose_random_flip,
    constraints=[single_flip_contiguous],
    accept=always_accept,
    initial_state=initial_partition,
    total_steps=1000
)
```

To configure a chain, we need to specify five objects.

proposal A function that takes the current state and returns new district assignments ("flips") for one or more nodes. This comes in the form of a dictionary mapping one or more node IDs to their new district IDs. Here we've used the `propose_random_flip` proposal, which proposes that a random node on the boundary of one district be flipped into the neighboring district.

constraints A list of binary constraints (functions that take a partition and return `True` or `False`) that together define which districting plans. are valid. Here we've used just a single constraint, `single_flip_contiguous`, which checks that each district in the plan is contiguous. This particular constraint is optimized for the single-flip proposal function we are using (hence the name). We could add more constraints to require that districts have nearly-equal population, to impose a bound on the compactness of the districts according to some score, or to prevent districts from splitting more counties than the original plan.

accept A function that takes a valid proposed state and returns `True` or `False` to signal whether the random walk should indeed move to the proposed state. `always_accept` always accepts valid proposed states. If you want to implement Metropolis-Hastings or any other more sophisticated acceptance criterion, you can specify your own custom acceptance function here.

initial_state The first state of the random walk.

total_steps The total number of steps to take. Invalid proposals are not counted toward this total, but rejected (by `accept`) valid states are.

For more information on the details of our Markov chain implementation, consult the *gerrychain.MarkovChain* documentation and source code.

The above code configures a Markov chain called `chain`, but does *not* run it yet. We run the chain by iterating through all of the states using a `for` loop. As an example, let's iterate through this chain and print out the sorted vector of

Democratic vote percentages in each district for each step in the chain.

```
for partition in chain:
    print(sorted(partition["SEN12"].percents("Dem")))
```

That's all: you've run a Markov chain!

To analyze the Republican vote percentages for each districting plan in our ensemble, we'll want to actually collect the data, and not just print it out. We can use a list comprehension to store these vote percentages, and then convert it into a `pandas DataFrame`.

```
import pandas

d_percents = [sorted(partition["SEN12"].percents("Dem")) for partition in chain]

data = pandas.DataFrame(d_percents)
```

This code will collect data from a different ensemble than our `for` loop above. Each time we iterate through the `chain` object, we run a fresh new Markov chain (using the same configuration that we defined when instantiating `chain`).

The pandas `DataFrame` object has many helpful methods for analyzing and plotting data. For example, we can produce a boxplot of our ensemble's Democratic vote percentage vectors, with the initial 2011 districting plan plotted in red, in just a few lines of code:

```
import matplotlib.pyplot as plt

ax = data.boxplot(positions=range(len(data.columns)))
data.iloc[0].plot(style="ro", ax=ax)

plt.show()
```

(Before you over-analyze this data, keep in mind that this is a toy ensemble of just one thousand plans created by single flips.)

2.1.4 Next steps

To see a more elaborate example that uses the ReCom proposal, see *Running a chain with ReCom*.

To learn more about the specific components of GerryChain, see the *API Reference*.

2.2 Running a chain with ReCom

This document shows how to run a chain using the ReCom proposal used in MGGG's 2018 Virginia House of Delegates report.

Our goal is to use ReCom to generate an ensemble of districting plans for Pennsylvania, and then make a box plot comparing the Democratic vote shares for plans in our ensemble to the 2011 districting plan that the Pennsylvania Supreme Court found to be a Republican-favoring partisan gerrymander.

This code is also available as a Jupyter notebook here.

2.2.1 Imports

The first step is to import everything we'll need:

```
import matplotlib.pyplot as plt
from gerrychain import (GeographicPartition, Partition, Graph, MarkovChain,
                        proposals, updaters, constraints, accept, Election)
from gerrychain.proposals import recom
from functools import partial
import pandas
```

2.2.2 Setting up the initial districting plan

Now we can load the adjacency graph of our state's VTDs. We created this graph from MGGG's Pennsylvania shapefile and saved it as a `.json` in *Getting started with GerryChain*.

```
graph = Graph.from_json("./PA_VTD.json")
```

We configure `Election` objects representing some of the election data from our shapefile.

```
elections = [
    Election("SEN10", {"Democratic": "SEN10D", "Republican": "SEN10R"}),
    Election("SEN12", {"Democratic": "USS12D", "Republican": "USS12R"}),
    Election("SEN16", {"Democratic": "T16SEND", "Republican": "T16SENR"}),
    Election("PRES12", {"Democratic": "PRES12D", "Republican": "PRES12R"}),
    Election("PRES16", {"Democratic": "T16PRESD", "Republican": "T16PRESR"})
]
```

Configuring our updaters

We want to set up updaters for everything we want to compute for each plan in the ensemble.

```
# Population updater, for computing how close to equality the district
# populations are. "TOT_POP" is the population column from our shapefile.
my_updaters = {"population": updaters.Tally("TOT_POP", alias="population")}

# Election updaters, for computing election results using the vote totals
# from our shapefile.
election_updaters = {election.name: election for election in elections}
my_updaters.update(election_updaters)
```

Instantiating the partition

We can now instantiate the initial state of our Markov chain, using the 2011 districting plan:

```
initial_partition = GeographicPartition(graph, assignment="2011_PLA_1", updaters=my_
↪updaters)
```

`GeographicPartition` comes with built-in `area` and `perimeter` updaters. We do not use them here, but they would allow us to compute compactness scores like Polsby-Popper that depend on these measurements.

2.2.3 Setting up the Markov chain

Proposal

First we'll set up the ReCom proposal. We need to fix some parameters using *functools.partial* before we can use it as our proposal function.

```
# The ReCom proposal needs to know the ideal population for the districts so that
# we can improve speed by bailing early on unbalanced partitions.

ideal_population = sum(initial_partition["population"].values()) / len(initial_
↪partition)

# We use functools.partial to bind the extra parameters (pop_col, pop_target, epsilon,
↪ node_repeats)
# of the recom proposal.
proposal = partial(recom,
                   pop_col="TOT_POP",
                   pop_target=ideal_population,
                   epsilon=0.02,
                   node_repeats=2
                  )
```

Constraints

To keep districts about as compact as the original plan, we bound the number of cut edges at 2 times the number of cut edges in the initial plan.

```
compactness_bound = constraints.UpperBound(
    lambda p: len(p["cut_edges"]),
    2*len(initial_partition["cut_edges"])
)

pop_constraint = constraints.within_percent_of_ideal_population(initial_partition, 0.
↪02)
```

Configuring the Markov chain

```
chain = MarkovChain(
    proposal=proposal,
    constraints=[
        pop_constraint,
        compactness_bound
    ],
    accept=accept.always_accept,
    initial_state=initial_partition,
    total_steps=1000
)
```

2.2.4 Running the chain

Now we'll run the chain, putting the sorted Democratic vote percentages directly into a `pandas DataFrame` for analysis and plotting. The `DataFrame` will have a row for each state of the chain. The first column of the `DataFrame` will hold the lowest Democratic vote share among the districts in each partition in the chain, the second column will hold the second-lowest Democratic vote shares, and so on.

```
# This will take about 10 minutes.

data = pandas.DataFrame(
    sorted(partition["SEN12"].percents("Democratic"))
    for partition in chain
)
```

If you install the `tqdm` package, you can see a progress bar as the chain runs by running this code instead:

```
data = pandas.DataFrame(
    sorted(partition["SEN12"].percents("Democratic"))
    for partition in chain.with_progress_bar()
)
```

2.2.5 Create a plot

Now we'll create a box plot similar to those appearing the Virginia report.

```
fig, ax = plt.subplots(figsize=(8, 6))

# Draw 50% line
ax.axhline(0.5, color="#cccccc")

# Draw boxplot
data.boxplot(ax=ax, positions=range(len(data.columns)))

# Draw initial plan's Democratic vote %s (.iloc[0] gives the first row)
data.iloc[0].plot(style="ro", ax=ax)

# Annotate
ax.set_title("Comparing the 2011 plan to an ensemble")
ax.set_ylabel("Democratic vote % (Senate 2012)")
ax.set_xlabel("Sorted districts")
ax.set_ylim(0, 1)
ax.set_yticks([0, 0.25, 0.5, 0.75, 1])

plt.show()
```

There you go! To build on this, here are some possible next steps:

- Add, remove, or tweak the constraints
- Use a different proposal from GerryChain, or create your own
- Perform a similar analysis on a different districting plan for Pennsylvania
- Perform a similar analysis on a different state
- Compute partisan symmetry scores like Efficiency Gap or Mean-Median, and create a histogram of the scores of the ensemble.
- Perform the same analysis using a different election than the 2012 Senate election
- Collect Democratic vote percentages for *all* the elections we set up, instead of just the 2012 Senate election.

We also highly recommend the resources prepared by Daryl R. DeFord of MGGG for the 2019 MIT IAP course Computational Approaches for Political Redistricting.

CHAPTER 3

API Reference

This document provides detailed documentation for classes and functions in GerryChain.

3.1 API Reference

Table of Contents

3.1.1 Adjacency graphs

class gerrychain.**Graph**(*incoming_graph_data=None*, ***attr*)

Represents a graph to be partitioned. It is based on `networkx.Graph`.

We have added some classmethods to help construct graphs from shapefiles, and to save and load graphs as JSON files.

Initialize a graph with edges, name, or graph attributes.

incoming_graph_data [input graph (optional, default: None)] Data to initialize graph. If None (default) an empty graph is created. The data can be an edge list, or any NetworkX graph object. If the corresponding optional Python packages are installed the data can also be a NumPy matrix or 2d ndarray, a SciPy sparse matrix, or a PyGraphviz graph.

attr [keyword arguments, optional (default= no attributes)] Attributes to add to graph as key=value pairs.

convert

```
>>> G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G = nx.Graph(name='my graph')
>>> e = [(1, 2), (2, 3), (3, 4)]  # list of edges
>>> G = nx.Graph(e)
```

Arbitrary graph attribute pairs (key=value) may be assigned

```
>>> G = nx.Graph(e, day="Friday")
>>> G.graph
{'day': 'Friday'}
```

add_data (*df*, *columns=None*)

Add columns of a DataFrame to a graph as node attributes using by matching the DataFrame's index to node ids.

Parameters

- **df** – Dataframe containing given columns.
- **columns** – (optional) List of dataframe column names to add.

classmethod from_file (*filename*, *adjacency='rook'*, *cols_to_add=None*, *reproject=True*, *ignore_errors=False*)

Create a *Graph* from a shapefile (or GeoPackage, or GeoJSON, or any other library that `geopandas` can read. See *from_geodataframe()* for more details.

Parameters **cols_to_add** – (optional) The names of the columns that you want to add to the graph as node attributes. By default, all columns are added.

classmethod from_geodataframe (*dataframe*, *adjacency='rook'*, *reproject=True*, *ignore_errors=False*)

Creates the adjacency *Graph* of geometries described by *dataframe*. The areas of the polygons are included as node attributes (with key *area*). The shared perimeter of neighboring polygons are included as edge attributes (with key *shared_perim*). Nodes corresponding to polygons on the boundary of the union of all the geometries (e.g., the state, if your dataframe describes VTDs) have a *boundary_node* attribute (set to *True*) and a *boundary_perim* attribute with the length of this "exterior" boundary.

By default, areas and lengths are computed in a UTM projection suitable for the geometries. This prevents the bizarro area and perimeter values that show up when you accidentally do computations in Longitude-Latitude coordinates. If the user specifies *reproject=False*, then the areas and lengths will be computed in the GeoDataFrame's current coordinate reference system. This option is for users who have a preferred CRS they would like to use.

Parameters

- **dataframe** – `geopandas.GeoDataFrame`
- **adjacency** – (optional) The adjacency type to use ("rook" or "queen"). Default is "rook".
- **reproject** – (optional) Whether to reproject to a UTM projection before creating the graph. Default is `True`.

- **ignore_errors** – (optional) Whether to ignore all invalid geometries and attept to create the graph anyway. Default is `False`.

Returns The adjacency graph of the geometries from *dataframe*.

Return type *Graph*

classmethod from_json(*json_file*)
Load a graph from a JSON file in the NetworkX json_graph format. :param json_file: Path to JSON file. :return: Graph

islands
The set of degree-0 nodes.

issue_warnings()
Issue warnings if the graph has any red flags (right now, only islands).

join(*dataframe*, *columns=None*, *left_index=None*, *right_index=None*)
Add data from a dataframe to the graph, matching nodes to rows when the node's *left_index* attribute equals the row's *right_index* value.

Parameters **dataframe** – DataFrame.

Columns (optional) The columns whose data you wish to add to the graph. If not provided, all columns are added.

Left_index (optional) The node attribute used to match nodes to rows. If not provided, node IDs are used.

Right_index (optional) The DataFrame column name to use to match rows to nodes. If not provided, the DataFrame's index is used.

to_json(*json_file*, *, *include_geometries_as_geojson=False*)
Save a graph to a JSON file in the NetworkX json_graph format. :param json_file: Path to target JSON file. :param bool include_geometry_as_geojson: (optional) Whether to include any

`shapely` geometry objects encountered in the graph's node attributes as GeoJSON. The default (`False`) behavior is to remove all geometry objects because they are not serializable. Including the GeoJSON will result in a much larger JSON file.

warn_for_islands()
Issue a warning if the graph has any islands (degree-0 nodes).

3.1.2 Partitions

class gerrychain.partition.**Partition**(*graph=None*, *assignment=None*, *updaters=None*, *parent=None*, *flips=None*)

Bases: `object`

Partition represents a partition of the nodes of the graph. It will perform the first layer of computations at each step in the Markov chain - basic aggregations and calculations that we want to optimize.

Parameters

- **graph** – Underlying graph; a NetworkX object.
- **assignment** – Dictionary assigning nodes to districts. If None, initialized to assign all nodes to district 0.
- **updaters** – Dictionary of functions to track data about the partition. The keys are stored as attributes on the partition class, which the functions compute.

crosses_parts(*edge*)
Answers the question "Does this edge cross from one part of the partition to another?

Parameters **edge** – tuple of node IDs

Return type bool

flip(*flips*)
Returns the new partition obtained by performing the given *flips* on this partition.

Parameters **flips** – dictionary assigning nodes of the graph to their new districts

Returns the new *Partition*

Return type *Partition*

classmethod from_districtr_file(*graph*, *districtr_file*, *updaters=None*)
Create a Partition from a districting plan created with Districtr, a free and open-source web app created by MGGG for drawing districts.

The provided `graph` should be created from the same shapefile as the Districtr module used to draw the districting plan. These shapefiles may be found in a repository in the mggg-states GitHub organization, or by request from MGGG.

Parameters

- **graph** – *Graph*
- **districtr_file** – the path to the `.json` file exported from Districtr
- **updaters** – dictionary of updaters

classmethod from_file(*filename*, *assignment*, *updaters=None*, *columns=None*)
Create a *Partition* from an ESRI Shapefile, a GeoPackage, a GeoJSON file, or any other file that the *fiona* library can handle.

classmethod from_json(*graph_path*, *assignment*, *updaters=None*)
Creates a *Partition* from a json file containing a serialized NetworkX *adjacency_data* object. Files of this kind for each state are available in the @gerrymandr/vtd-adjacency-graphs GitHub repository.

Parameters

- **graph_path** – String filename for the json file
- **assignment** – String key for the node attribute giving a district assignment, or a dictionary mapping node IDs to district IDs.
- **updaters** – (optional) Dictionary of updater functions to attach to the partition, in addition to the default_updaters of *cls*.

plot(*geometries*, ***kwargs*)
Plot the partition, using the provided geometries.

Parameters

- **geometries** – A `geopandas.GeoDataFrame` or `geopandas.GeoSeries` holding the geometries to use for plotting. Its `Index` should match the node labels of the partition's underlying *Graph*.
- ****kwargs** – Additional arguments to pass to `geopandas.GeoDataFrame.plot()` to adjust the plot.

to_json(*json_path*, ***, *save_assignment_as=None*, *include_geometries_as_geojson=False*)
Save the partition to a JSON file in the NetworkX json_graph format.

Parameters

- **json_file** – Path to target JSON file.
- **save_assignment_as** (*str*) – (optional) The string to use as a node attribute key holding the current assignment. By default, does not save the assignment as an attribute.
- **include_geometries_as_geojson** (*bool*) – (optional) Whether to include any `shapely` geometry objects encountered in the graph's node attributes as GeoJSON. The default (`False`) behavior is to remove all geometry objects because they are not serializable. Including the GeoJSON will result in a much larger JSON file.

class gerrychain.partition.**GeographicPartition**(*graph=None*, *assignment=None*, *updaters=None*, *parent=None*, *flips=None*)

Bases: gerrychain.partition.partition.Partition

A *Partition* with areas, perimeters, and boundary information included. These additional data allow you to compute compactness scores like Polsby-Popper.

Parameters

- **graph** – Underlying graph; a NetworkX object.
- **assignment** – Dictionary assigning nodes to districts. If None, initialized to assign all nodes to district 0.
- **updaters** – Dictionary of functions to track data about the partition. The keys are stored as attributes on the partition class, which the functions compute.

3.1.3 Markov chains

class gerrychain.**MarkovChain**(*proposal*, *constraints*, *accept*, *initial_state*, *total_steps*)

MarkovChain is an iterator that allows the user to iterate over the states of a Markov chain run.

Example usage:

```
chain = MarkovChain(proposal, constraints, accept, initial_state, total_steps)
for state in chain:
    # Do whatever you want - print output, compute scores, ...
```

Parameters

- **proposal** – Function proposing the next state from the current state.
- **constraints** – A function with signature `Partition -> bool` determining whether the proposed next state is valid (passes all binary constraints). Usually this is a *Validator* class instance.
- **accept** – Function accepting or rejecting the proposed state. In the most basic use case, this always returns `True`. But if the user wanted to use a Metropolis-Hastings acceptance rule, this is where you would implement it.
- **initial_state** – Initial *gerrychain.partition.Partition* class.
- **total_steps** – Number of steps to run.

3.1.4 Proposals

3.1.5 Binary constraints

The *gerrychain.constraints* module provides a collection of constraint functions and helper classes for the validation step in GerryChain.

Helper classes	
`Validator`	Collection of constraints
`Bounds`	Bounds on numeric constraints
`UpperBounds`	Upper bounds on numeric constraints
`LowerBounds`	Lower bounds on numeric constraints
`SelfConfiguringUpperBound`	Automatic upper bounds on numeric constraints
`SelfConfiguringLowerBound`	Automatic lower bounds on numeric constraints
`WithinPercentRangeOfBounds`	Percentage bounds for numeric constraints

Binary constraint functions	
`no_worse_L1_reciprocal_polsby_popper`	Lower bounded L1-reciprocal Polsby-Popper
`no_worse_L_minus_1_reciprocal_polsby_popper`	Lower bounded L(-1)-reciprocal Polsby-Popper
`contiguous`	Districts are contiguous (with NetworkX methods)
`contiguous_bf`	Districts are contiguous (with a breadth-first search)
`single_flip_contiguous`	Districts are contiguous (optimized for `propose_random_flip` proposal)
`no_vanishing_districts`	No districts may be completely consumed

Each new step proposed to the chain is passed off to the "validator" functions here to determine whether or not the step is valid. If it is invalid (breaks contiguity, for instance), then the step is immediately rejected.

A validator should take in a *Partition* instance, and should return whether or not the instance is valid according to their rules. Many top-level functions following this signature in this module are examples of this.

class gerrychain.constraints.**Validator**(*constraints*)

A single callable for checking that a partition passes a collection of constraints. Intended to be passed as the `is_valid` parameter when instantiating *MarkovChain*.

This class is meant to be called as a function after instantiation; its return is `True` if all validators pass, and `False` if any one fails.

Example usage:

```
is_valid = Validator([constraint1, constraint2, constraint3])
chain = MarkovChain(proposal, is_valid, accept, initial_state, total_steps)
```

Parameters **constraints** – List of validator functions that will check partitions.

3.1.6 Updaters

gerrychain.updaters.**county_splits**(*partition_name*, *county_field_name*)

Track county splits.

Parameters

- **partition_name** – Name that the *Partition* instance will store.
- **county_field_name** – Name of county ID field on the graph.

Returns The tracked data is a dictionary keyed on the county ID. The stored values are tuples of the form *(split, nodes, seen)*. *split* is a *CountySplit* enum, *nodes* is a list of node IDs, and *seen* is a list of assignment IDs that are contained in the county.

class gerrychain.updaters.**Tally**(*fields*, *alias=None*, *dtype=<class 'int'>*)
An updater for keeping a tally of one or more node attributes.

Parameters

- **fields** – the list of node attributes that you want to tally. Or a just a single attribute name as a string.
- **alias** – the aliased name of this Tally (meaning, the key corresponding to this Tally in the Partition's updaters dictionary)
- **dtype** – the type (int, float, etc.) that you want the tally to have

class gerrychain.updaters.**DataTally**(*data*, *alias*)
An updater for tallying numerical data that is not necessarily stored as node attributes

Parameters

- **data** – a dict or Series indexed by the graph's nodes, or the string key for a node attribute containing the Tally's data.
- **alias** – the name of the tally in the Partition's *updaters* dictionary

class gerrychain.updaters.**CountySplit**
An enumeration.

class gerrychain.updaters.**Election**(*name*, *parties_to_columns*, *alias=None*)
Represents the data of one election, with races conducted in each part of the partition.

As we vary the districting plan, we can use the same node-level vote totals to tabulate hypothetical elections. To do this manually with tallies, we would have to maintain tallies for each party, as well as the total number of votes, and then compute the electoral results and percentages from scratch every time. To make this simpler, this class provides an `ElectionUpdater` to manage these tallies. The updater returns an `ElectionResults` class giving a convenient view of the election results, with methods like `wins()` or `percent()` for common queries the user might make on election results.

Example usage:

```
# Assuming your nodes have attributes "2008_D", "2008_R"
# with (for example) 2008 senate election vote totals
election = Election(
    "2008 Senate",
    {"Democratic": "2008_D", "Republican": "2008_R"},
    alias="2008_Sen"
)

# Assuming you already have a graph and assignment:
partition = Partition(
    graph,
    assignment,
    updaters={"2008_Sen": election}
)
```

(continues on next page)

(continued from previous page)

```
# The updater returns an ElectionResults instance, which
# we can use (for example) to see how many seats a given
# party would win in this partition using this election's
# vote distribution:
partition["2008_Sen"].wins("Republican")
```

Parameters

- **name** – The name of the election. (e.g. "2008 Presidential")
- **parties_to_columns** – A dictionary matching party names to their data columns, either as actual columns (list-like, indexed by nodes) or as string keys for the node attributes that hold the party's vote totals. Or, a list of strings which will serve as both the party names and the node attribute keys.
- **alias** – (optional) Alias that the election is registered under in the Partition's dictionary of updaters.

3.1.7 Elections

class gerrychain.updaters.election.**Election**(*name*, *parties_to_columns*, *alias=None*)

Represents the data of one election, with races conducted in each part of the partition.

As we vary the districting plan, we can use the same node-level vote totals to tabulate hypothetical elections. To do this manually with tallies, we would have to maintain tallies for each party, as well as the total number of votes, and then compute the electoral results and percentages from scratch every time. To make this simpler, this class provides an *ElectionUpdater* to manage these tallies. The updater returns an *ElectionResults* class giving a convenient view of the election results, with methods like *wins()* or *percent()* for common queries the user might make on election results.

Example usage:

```
# Assuming your nodes have attributes "2008_D", "2008_R"
# with (for example) 2008 senate election vote totals
election = Election(
    "2008 Senate",
    {"Democratic": "2008_D", "Republican": "2008_R"},
    alias="2008_Sen"
)

# Assuming you already have a graph and assignment:
partition = Partition(
    graph,
    assignment,
    updaters={"2008_Sen": election}
)

# The updater returns an ElectionResults instance, which
# we can use (for example) to see how many seats a given
# party would win in this partition using this election's
# vote distribution:
partition["2008_Sen"].wins("Republican")
```

Parameters

- **name** – The name of the election. (e.g. "2008 Presidential")

- **parties_to_columns** – A dictionary matching party names to their data columns, either as actual columns (list-like, indexed by nodes) or as string keys for the node attributes that hold the party's vote totals. Or, a list of strings which will serve as both the party names and the node attribute keys.
- **alias** – (optional) Alias that the election is registered under in the Partition's dictionary of updaters.

class gerrychain.updaters.election.**ElectionResults**(*election*, *counts*, *races*)

Represents the results of an election. Provides helpful methods to answer common questions you might have about an election (Who won? How many seats?, etc.).

count(*party*, *race=None*)

Returns the total number of votes that `party` received in a given race (part of the partition). If `race` is omitted, returns the overall vote total of `party`.

Parameters

- **party** – Party ID.
- **race** – ID of the part of the partition whose votes we want to tally.

counts(*party*)

Parameters **party** – Party ID

Returns tuple of the total votes cast for `party` in each part of the partition

percent(*party*, *race=None*)

Returns the percentage of the vote that `party` received in a given race (part of the partition). If `race` is omitted, returns the overall vote share of `party`.

Parameters

- **party** – Party ID.
- **race** – ID of the part of the partition whose votes we want to tally.

percents(*party*)

Parameters **party** – The party

Returns The tuple of the percentage of votes that `party` received in each part of the partition

seats(*party*)

Returns the number of races that `party` won.

votes(*party*)

An alias for *counts()*.

wins(*party*)

An alias for *seats()*.

won(*party*, *race*)

Answers "Did `party` win the race in part `race`?" with `True` or `False`.

class gerrychain.updaters.election.**ElectionUpdater**(*election*)

The updater for computing the election results in each part of the partition after each step in the Markov chain. The actual results are returned to the user as an *ElectionResults* instance.

3.1.8 Grids

To make it easier to play around with GerryChain, we have provided a *Grid* class representing a partition of a grid graph. This is especially useful if you want to start experimenting but do not yet have a clean set of data and geometries to build your graph from.

class gerrychain.grid.**Grid**(*dimensions=None*, *with_diagonals=False*, *assignment=None*, *updaters=None*, *parent=None*, *flips=None*)

The *Grid* class represents a grid partitioned into districts. It is useful for running little experiments with GerryChain without needing to do any data processing or cleaning to get started.

Example usage:

```
grid = Grid((10,10))
```

The nodes of `grid.graph` are labelled by tuples `(i,j)`, for `0 <= i <= 10` and `0 <= j <= 10`. Each node has an `area` of 1 and each edge has `shared_perim` 1.

Parameters

- **dimensions** – tuple (m,n) of the desired dimensions of the grid.
- **with_diagonals** – (optional, defaults to False) whether to include diagonals as edges of the graph (i.e., whether to use 'queen' adjacency rather than 'rook' adjacency).
- **assignment** – (optional) dict matching nodes to their districts. If not provided, partitions the grid into 4 quarters of roughly equal size.
- **updaters** – (optional) dict matching names of attributes of the Partition to functions that compute their values. If not provided, the Grid configures the cut_edges updater for convenience.

as_list_of_lists()

Returns the grid as a list of lists (like a matrix), where the (i,j)th entry is the assigned district of the node in position (i,j) on the grid.

3.1.9 Spanning tree methods

The `recom()` proposal function operates on spanning trees of the adjacency graph in order to generate new contiguous districting plans with balanced population.

The *gerrychain.tree* submodule exposes some helpful functions for partitioning graphs using spanning trees methods. These may be used to implement proposal functions or to generate initial plans for running Markov chains, as described in MGGG's 2018 Virginia House of Delegates report.

gerrychain.tree.**bipartition_tree**(*graph*, *pop_col*, *pop_target*, *epsilon*, *node_repeats*, *spanning_tree=None*, *choice=<bound method Random.choice of <random.Random object>>*)

This function finds a balanced 2 partition of a graph by drawing a spanning tree and finding an edge to cut that leaves at most an epsilon imbalance between the populations of the parts. If a root fails, new roots are tried until node_repeats in which case a new tree is drawn.

Builds up a connected subgraph with a connected complement whose population is `epsilon * pop_target` away from `pop_target`.

Returns a subset of nodes of `graph` (whose induced subgraph is connected). The other part of the partition is the complement of this subset.

Parameters

- **graph** – The graph to partition

- **pop_col** – The node attribute holding the population of each node
- **pop_target** – The target population for the returned subset of nodes
- **epsilon** – The allowable deviation from `pop_target` (as a percentage of `pop_target`) for the subgraph's population
- **node_repeats** – A parameter for the algorithm: how many different choices of root to use before drawing a new spanning tree.
- **spanning_tree** – The spanning tree for the algorithm to use (used when the algorithm chooses a new root and for testing)
- **choice** – `random.choice()`. Can be substituted for testing.

gerrychain.tree.**recursive_tree_part**(*graph*, *parts*, *pop_target*, *pop_col*, *epsilon*, *node_repeats=None*)

Uses `bipartition_tree()` recursively to partition a tree into `len(parts)` parts of population `pop_target` (within `epsilon`). Can be used to generate initial seed plans or to implement ReCom-like "merge walk" proposals.

Parameters

- **graph** – The graph
- **parts** – Iterable of part labels (like `[0,1,2]` or `range(4)`
- **pop_target** – Target population for each part of the partition
- **pop_col** – Node attribute key holding population data
- **epsilon** – How far (as a percentage of `pop_target`) from `pop_target` the parts of the partition can be
- **node_repeats** – Parameter for `bipartition_tree()` to use.

Returns New assignments for the nodes of `graph`.

Return type dict

CHAPTER 4

Topics

4.1 Reproducibility

If you've used GerryChain to do some analysis or research, you may want to ensure that your analysis is completely repeatable by anyone else on their own computer. This guide will walk you through the steps required to make that possible.

4.1.1 Share your code on GitHub

Before anyone can run your code, they'll need to find it. We strongly recommend publishing your source code as a GitHub repository, and not as a `.zip` file on your personal website. GitHub has a desktop client that makes this easy.

4.1.2 Use the same versions of all of your dependencies

You will want to make sure that anyone who tries to repeat your analysis by running your code will have the exact same versions of all of the software and packages that you use, including the same version of Python.

The easiest way to do this is to use conda to manage all of your dependencies. You can use conda to export an `environment.yml` file that anyone can use to replicate your environment by running the command `conda env create -f environment.yml`. For instructions on how to do this, see Sharing your environment and Creating an environment from an environment.yml file in the conda documentation.

If you've published your code on GitHub, it is a good idea to include your `environment.yml` file in the root folder of your code repository.

4.1.3 Import `random` from `gerrychain.random`

The submodule `gerrychain.random` is the single place where GerryChain imports the built-in Python module `random` and sets a random seed. This makes sure that all randomness is used *after* the seed is set. If you use the `random` module anywhere in your own code (say, in your own proposal function), replace the line `import`

`random` with `from gerrychain.random import random`. This will ensure that your code uses the same random seed as GerryChain.

GerryChain sets a random seed of `2018` after it imports `random`. If you wish to use a different random seed, set it immediately after importing `random` from `gerrychain.random`, and *before* you import anything else. That will look like this:

```
from gerrychain.random import random
random.seed(12345678)

from gerrychain import MarkovChain, Partition
# and so on...
```

4.1.4 Set `PYTHONHASHSEED=0`

In addition to the randomness provided by the `random` module, Python uses a random seed for its hashing algorithm, which affects how objects are stored in sets and dictionaries. This must happen the same way every time in order for GerryChain runs to be repeatable.

The way to accomplish this is to set the environment variable `PYTHONHASHSEED` to `0`.

If you are using conda for managing packages, dependencies, and environments, you can save environment variables in your conda environment.

Otherwise, in macOS or Linux environments you can accomplish this by running the command `export PYTHONHASHSEED=0` in the Terminal or bash shell before running your code.

In a Windows 10 environment using PowerShell, you can accomplish this by running `$env:PYTHONHASHSEED=0` before running your code.

Python Module Index

g

Index

S

T

V

W