GerryChain Documentation

Metric Geometry and Gerrymandering Group

Feb 04, 2020

User Guide

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 -n base 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.

1.2.1 Installation

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 -n base conda
conda init
```

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

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.

1.2.2 Getting started with GerryChain

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

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 MGGG's json 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_VTDs.json` file that you downloaded.

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_json("./PA_VTDs.json")

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

initial_partition = Partition(
```

(continues on next page)

(continued from previous page)

```
    graph,
    assignment="CD_2011",
    updaters={
        "cut_edges": cut_edges,
        "population": Tally("TOTPOP", alias="population"),
        "SEN12": election
    }
)
```

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

The *Graph.from_json()* classmethod creates a *Graph* of the precincts. 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="CD_2011"` to tell the `Partition` to assign nodes by their `"CD_2011"` 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 `"TOTPOP"` 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.

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)))
plt.plot(data.iloc[0], "ro")

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.)

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*.

1.2.3 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.

You can run this example in an interactive Jupyter Notebook session in your browser, without installing anything, using Binder:

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
```

Setting up the initial districting plan

We'll create our graph using the Pennsylvania shapefile from MGGG-States (download the .json here if you didn't start with the Getting started with GerryChain guide):

```
graph = Graph.from_json("./PA_VTDs.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. "TOTPOP" is the population column from our shapefile.
my_updaters = {"population": updaters.Tally("TOTPOP", 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="CD_2011", 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.

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="TOTPOP",
                   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
)
```

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()
)
```

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)
plt.plot(data.iloc[0], "ro")

# 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.

1.2.4 Working with Partitions

This document walks you through the most common ways that you might work with a GerryChain `Partition` object.

```
>>> import geopandas
>>> from gerrychain import Partition, Graph
>>> from gerrychain.updaters import cut_edges
```

We'll use our Pennsylvania VTD json to create the graph we'll use in these examples.

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

Creating a partition

Here is how you can create a Partition:

```
>>> partition = Partition(graph, "CD_2011", {"cut_edges": cut_edges})
```

The `Partition` class takes three arguments to create a Partition:

- A **graph**.
- An **assignment of nodes to districts**. This can be the string name of a node attribute (shapefile column) that holds each node's district assignment, or a dictionary mapping each node ID to its assigned district ID.
- A dictionary of **updaters**.

This creates a partition of the `graph` object we created above from the Pennsylvania shapefile. The partition is defined by the `"CD_2011"` column from our shapefile's attribute table.

`partition.graph`: the underlying graph

You can access the partition's underlying Graph as `partition.graph`. This contains no information about the partition—it will be the same graph object that you passed in to `Partition()` when you created the partition instance.

`partition.graph` is a `gerrychain.Graph` object. It is based on the NetworkX Graph object, so any functions (e.g. `connected_components`) you can find in the NetworkX documentation will be compatible.

```
>>> partition.graph
<Graph [8921 nodes, 25228 edges]>
```

Now we have a graph of Pennsylvania's VTDs, with all of the data from our shapefile's attribute table attached to the graph as *node attributes*. We can see the data that a node has like this:

```
>>> partition.graph.nodes[0]
{'boundary_node': True,
 'boundary_perim': 0.06312599142331599,
 'area': 0.004278359631999892,
 'STATEFP10': '42',
```

(continues on next page)

(continued from previous page)

```
'COUNTYFP10': '085',
'VTDST10': '960',
'GEOID10': '42085960',
'VTDI10': 'A',
'NAME10': 'SHENANGO TWP VTD WEST',
'NAMELSAD10': 'SHENANGO TWP VTD WEST',
'LSAD10': '00',
'MTFCC10': 'G5240',
'FUNCSTAT10': 'N',
'ALAND10': 39740056,
'AWATER10': 141805,
'INTPTLAT10': '+41.1564874',
'INTPTLON10': '-080.4865792',
'TOTPOP': 1915,
'NH_WHITE': 1839,
'NH_BLACK': 35'
'NH_AMIN': 1,
'NH_ASIAN': 8,
'NH_NHPI': 0,
'NH_OTHER': 3,
'NH_2MORE': 19,
'HISP': 10,
'H_WHITE': 3,
'H_BLACK': 0,
'H_AMIN': 1,
'H_ASIAN': 0,
'H_NHPI': 0,
'H_OTHER': 4,
'H_2MORE': 2,
'VAP': 1553,
'HVAP': 7,
'WVAP': 1494,
'BVAP': 30,
'AMINVAP': 1,
'ASIANVAP': 6,
'NHPIVAP': 0,
'OTHERVAP': 2,
'2MOREVAP': 13,
'ATG12D': 514.0001036045286,
'ATG12R': 388.0000782073095,
'F2014GOVD': 290.0000584539169,
'F2014GOVR': 242.00004877878584,
'GOV10D': 289.00005825235166,
'GOV10R': 349.00007034626555,
'PRES12D': 492.0000991700935,
'PRES12O': 11.000002217217538,
'PRES12R': 451.0000909059191,
'SEN10D': 315.00006349304766,
'SEN10R': 328.0000661133957,
'T16ATGD': 416.00008385113597,
'T16ATGR': 558.0001124733988,
'T16PRESD': 342.0000689353089,
'T16PRESOTH': 32.00000645008738,
'T16PRESR': 631.0001271876606,
'T16SEND': 379.00007639322246,
'T16SENR': 590.0001189234862,
'USS12D': 505.00010179044153,
```

(continues on next page)

(continued from previous page)

```
 'USS12R': 423.0000852620926,
 'REMEDIAL': '16',
 'GOV': '3',
 'TS': 3,
 'CD_2011': 3,
 'SEND': 50,
 'HDIST': 7,
 '538DEM': '03',
 '538GOP': '03',
 '538CMPCT': '03',
 'geometry': <shapely.geometry.polygon.Polygon object at 0x7ff3edeb7f90>}
```

The nodes of the graph are identified by IDs. Here the IDs are the VTDs GEOIDs from the `"GEOID10"` column from our shapefile.

`partition.assignment`: assign nodes to parts

`partition.assignment` gives you a mapping from node IDs to part IDs ("part" is our generic word for "district"). It is a custom data structure but you can use it just like a dictionary.

```
>>> first_ten_nodes = list(partition.graph.nodes)[:10]
>>> for node in first_ten_nodes:
...     print(partition.assignment[node])
3
3
3
3
3
3
3
3
3
3
```

`partition.parts`: the nodes in each part

`partition.parts` gives you a mapping from each part ID to the set of nodes that belong to that part. This is the "opposite" mapping of `assignment`.

As an example, let's print out the number of nodes in each part:

```
>>> for part in partition.parts:
...     number_of_nodes = len(partition.parts[part])
...     print(f"Part {part} has {number_of_nodes} nodes")
Part 3 has 500 nodes
Part 5 has 580 nodes
Part 10 has 515 nodes
Part 9 has 575 nodes
Part 12 has 623 nodes
Part 6 has 313 nodes
Part 15 has 324 nodes
Part 7 has 405 nodes
Part 16 has 329 nodes
Part 11 has 456 nodes
```

(continues on next page)

(continued from previous page)

```
Part 4 has 292 nodes
Part 8 has 340 nodes
Part 17 has 442 nodes
Part 18 has 600 nodes
Part 14 has 867 nodes
Part 13 has 548 nodes
Part 2 has 828 nodes
Part 1 has 718 nodes
```

Notice that `partition.parts` might not loop through the parts in numerical order—but it will always loop through the parts in the same order. (You can run the cell above multiple times to verify that the order doesn't change.)

`partition.subgraphs`: the subgraphs of each part

For each part of our partition, we can look at the *subgraph* that it defines. That is, we can look at the graph made up of all the nodes in a certain part and all the edges between those nodes.

`partition.subgraphs` gives us a mapping (like a dictionary) from part IDs to their subgraphs. These subgraphs are NetworkX Subgraph objects, and work exactly like our main graph object—nodes, edges, and node attributes all work the same way.

```
>>> for part, subgraph in partition.subgraphs.items():
...     number_of_edges = len(subgraph.edges)
...     print(f"Part {part} has {number_of_edges} edges")
Part 3 has 1229 edges
Part 5 has 1450 edges
Part 10 has 1252 edges
Part 9 has 1391 edges
Part 12 has 1601 edges
Part 6 has 749 edges
Part 15 has 834 edges
Part 7 has 931 edges
Part 16 has 836 edges
Part 11 has 1152 edges
Part 4 has 723 edges
Part 8 has 886 edges
Part 17 has 1092 edges
Part 18 has 1585 edges
Part 14 has 2344 edges
Part 13 has 1362 edges
Part 2 has 2159 edges
Part 1 has 1780 edges
```

Let's use NetworkX's diameter function to compute the diameter of each part subgraph. (The *diameter* of a graph is the length of the longest shortest path between any two nodes in the graph. You don't have to know that!)

```
>>> import networkx
>>> for part, subgraph in partition.subgraphs.items():
...     diameter = networkx.diameter(subgraph)
...     print(f"Part {part} has diameter {diameter}")
Part 3 has diameter 40
Part 5 has diameter 30
Part 10 has diameter 40
Part 9 has diameter 40
Part 12 has diameter 36
```

(continues on next page)

(continued from previous page)

```
Part 6 has diameter 32
Part 15 has diameter 28
Part 7 has diameter 38
Part 16 has diameter 38
Part 11 has diameter 31
Part 4 has diameter 19
Part 8 has diameter 24
Part 17 has diameter 34
Part 18 has diameter 28
Part 14 has diameter 38
Part 13 has diameter 30
Part 2 has diameter 28
Part 1 has diameter 50
```

Outputs of updaters

The other main way we can extract information from `partition` is through the updaters that we configured when we created it. We gave `partition` just one updater, `cut_edges`. This is the set of edges that go between nodes that are in *different* parts of the partition.

```
>>> len(partition["cut_edges"])
2367
>>> len(partition.cut_edges)
2367
```

```
>>> proportion_of_cut_edges = len(partition.cut_edges) / len(partition.graph.edges)
>>> print("Proportion of edges that are cut:")
>>> print(proportion_of_cut_edges)
Proportion of edges that are cut:
0.09201881584574116
```

1.2.5 Updaters

Depending on the questions you are investigating, there are many different values you might want to compute for each partition in your Markov chain. If you are interested in compactness, you might want to compute the area and perimeter of each part of the partition so that you can compute compactness scores. If you are interested in partisan lean, you might want to compute hypothetical election results using the districts defined by the partition.

The `Partition` class allows you to define custom properties for the partitions in your Markov chain. You can do this by providing a dictionary of updater functions when you first create a partition.

```
>>> from gerrychain import Partition, Graph
>>>
>>> graph = Graph()
>>> graph.add_edges_from([(0, 1), (1, 2), (2, 0)])
>>> assignment = {0: 1, 1: 1, 2: 2}
>>>
>>> def my_updater(partition):
...     return "Hello!"
>>>
>>> partition = Partition(graph, assignment, {"my_custom_property": my_updater})
>>> partition["my_custom_property"]
'Hello!'
```

(continues on next page)

(continued from previous page)

```
>>> partition.my_custom_property
'Hello!'
```

As shown in this example, you can access the value of this updater using either the "attribute-style" syntax `partition.my_custom_property` or the "dictionary-style" syntax `partition["my_custom_property"]`.

This partition and all subsequent partitions in the chain will have this `my_custom_property` attribute. If we flip a node in `partition` to create a new partition, we can still access this property:

```
>>> new_partition = partition.flip({1: 2})
>>> new_partition is not partition
True
>>> new_partition["my_custom_property"]
'Hello!'
```

Useful updater functions in GerryChain

The `gerrychain.updaters` submodule provides some updaters for common tasks like aggregating data and computing the cut edges of a partition:

- `Tally`: Aggregates a node attribute (e.g. population) over each part of the partition.
- `cut_edges`: Returns the set of cut edges (edges whose nodes are in different parts of the partition) of the partition. This is required for most of the proposal functions in `gerrychain.proposals`.

Here is an example using both of these updaters:

```
>>> from gerrychain.updaters import cut_edges, Tally
>>> # We'll use a 2x2 grid graph:
>>> graph = Graph()
>>> graph.add_edges_from([(0, 1), (1, 2), (2, 3), (3, 0)])
>>> # Give each of the nodes population 100:
>>> for node in graph:
...     graph.nodes[node]["population"] = 100
>>>
>>> # Partition the grid into two halves:
>>> assignment = {0: 0, 1: 0, 2: 1, 3: 1}
>>> partition = Partition(
...     graph,
...     assignment,
...     updaters={"cut_edges": cut_edges, "population": Tally("population")}
... )
>>> partition["population"]
{0: 200, 1: 200}
>>> partition["cut_edges"]
{(1, 2), (0, 3)}
```

Our `cut_edges` updater returns a set of edges, each represented as a tuple of two nodes. Our `population` updater returns a dictionary mapping each part of the partition to the total population in that part. Since we divided our grid in half, we see parts `0` and `1` both have population 200.

Now when we create a new partition by flipping a node of `partition`, we see the values of the updaters change:

```
>>> new_partition = partition.flip({0: 1})
>>> new_partition["population"]
```

(continues on next page)

(continued from previous page)

```
{0: 100, 1: 300}
>>> new_partition["cut_edges"]
{(1, 2), (0, 1)}
```

As we should expect, flipping node `0` into part `1` increases the population of part `1` to 300 and decreases the population of part `0` to 100. The cut edges of the new partition are both of the edges incident to node `1`, since this is the last remaining node in part `0`.

Writing your own updater function

When using GerryChain to experiment with new metrics, proposals, or acceptance rules, there usually comes a point when you need to implement a new updater. As we saw in the first example, an updater is a function that takes the partition as its argument and returns any type of value.

Let's create an updater that returns the number of cut edges in the partition.

```
>>> def number_of_cut_edges(partition):
...     return len(partition["cut_edges"])
```

Note that this updater uses the value of the `cut_edges` updater in its computation. This is completely allowed! All you need to do is make sure that any updater that your updater depends on is included in the `updaters` dictionary that we pass to `Partition`. We also need to make sure that we have no cyclic depedencies: if the `cut_edges` updater also depended on `number_of_cut_edges`, we would fall into an infinite loop when we called either of them, resulting in a `RuntimeError: maximum recursion depth exceeded` error.

To try out our updater, we'll use NetworkX to create a complete graph on 4 nodes, which we'll partition in halves like the 2x2 grid. NetworkX is the graph library that GerryChain uses under the hood in its `Graph` class.

```
>>> import networkx
>>> graph = Graph(networkx.complete_graph(4))
>>> assignment = {0: 0, 1: 0, 2: 1, 3: 1}
>>> my_updaters = {"cut_edges": cut_edges, "number_of_cut_edges": number_of_cut_edges}
>>> partition = Partition(graph, assignment, my_updaters)
```

Now we can try out our custom `number_of_cut_edges` updater, and verify that its value changes when the partition changes:

```
>>> partition["number_of_cut_edges"]
4
>>> new_partition = partition.flip({0: 1})
>>> new_partition["number_of_cut_edges"]
3
```

1.2.6 What to do about islands and connectivity

When you create or load a graph, you may have see a warning about islands, like this one for Massachusetts:

```
>>> from gerrychain import Graph
>>> graph = Graph.from_json("./Block_Groups/BG25.json")
graph.py:216: UserWarning: Found islands (degree-0 nodes). Indices of islands: {2552,␣
↪3107}
    "Found islands (degree-0 nodes). Indices of islands: {}".format(islands)
```

This warning is telling you that there are a couple nodes (2552, 3107) that have no neighbors—that is, they are completely disconnected from the rest of the graph. In addition to these nodes without neighbors, we can also have a whole cluster of nodes that is disconnected from the rest of the graph.

These are a problem because we want the districting plans we generate to have contiguous districts, and any district with one of these islands will not be contiguous. In addition, spanning tree-based methods like the ReCom proposal or the `recursive_tree_part` seed-creation method require that the graph be connected—starting with a disconnected graph can lead to mysterious `KeyErrors` and infinite loops.

These islands and disconnected components show up when there are actual geographic islands, or (for example) when there is a unit that is connected to the mainland by a bridge but whose official boundary in the shapefile does not touch the mainland's boundary.

This notebook will show you how to diagnose these contiguity problems, and will give you a few different strategies for dealing with islands and disconnected components so that you can get your graph to a workable state for generating ensembles of districting plans.

Figuring out what is going on

The first step in the process is to figure out what is going on. To see the node IDs of the islands that GerryChain warned us about, we can inspect `graph.islands`:

```
>>> islands = graph.islands
>>> islands
{2552, 3107}
```

GerryChain warns you when there are islands, but we'll also want to see if there are whole disconnected components of the graph. We can use the NetworkX function `connected_components` to do this. The `connected_components` function yields the set of node IDs in each connected component of the graph. We'll put these sets into a list, and then print out their lengths to see how bad the contiguity situation is:

```
>>> from networkx import is_connected, connected_components
>>> components = list(connected_components(graph))
>>> [len(c) for c in components]
[4922, 25, 19, 11, 1, 1]
```

We have one big component with 4922 nodes. The two components with 1 node are the islands that GerryChain warned us about. Then we also have another few disconnected components with 25, 19, and 11 nodes each.

To fix these problems, we have two options. The first strategy is to just delete these disconnected components, and do our analysis on the big connected component. This might be the right way to go if you are just trying to get started—you can always go back and make a better fix later when you want to make sure your results are sound.

The second strategy is to manually add edges to the graph to connect all of the disconnected components. This is more involved but will give you a much better graph for your analysis.

Strategy 1: delete the problem components

For the quick and very dirty route, we can just delete all of the nodes in components that are causing us problems:

```
>>> biggest_component_size = max(len(c) for c in components)
>>> problem_components = [c for c in components if len(c) != biggest_component_size]
>>> for component in problem_components:
...     for node in component:
...         graph.remove_node(node)
```

We can verify that our graph is now connected:

```
>>> is_connected(graph)
True
```

And now we're ready to run a chain! Let's save the graph in a new `.json` file so that we don't have to do this fix every time we want to run a chain.

```
>>> graph.to_json("./my_graph_with_islands_deleted.json")
```

Strategy 2: connect the components manually

Deleting the problem components will make it easier for your code to run, but we should not use this method when we actually care about the results we are getting. Deleting nodes from the graph also deletes all the people who live in those nodes, which is strictly not OK, morally speaking. For a better, more laborious solution, we will:

- Use QGIS to open the Shapefile that our graph was made from
- Examine the disconnected components in QGIS, and
- Judiciously add edges to the graph to make it connected.

This requires making some human judgements about which units are "near" to each other, for the purposes of district contiguity. This is an art more than a science, with a lot of questions with no single right answer. For instance, should all of the Hawaiian islands be connected to one another, or connected in a chain from east to west? Should the upper peninsula of Michigan connect to the lower pensinsula only at the Mackinac bridge, or all along the coast?

Downloading the shapefile

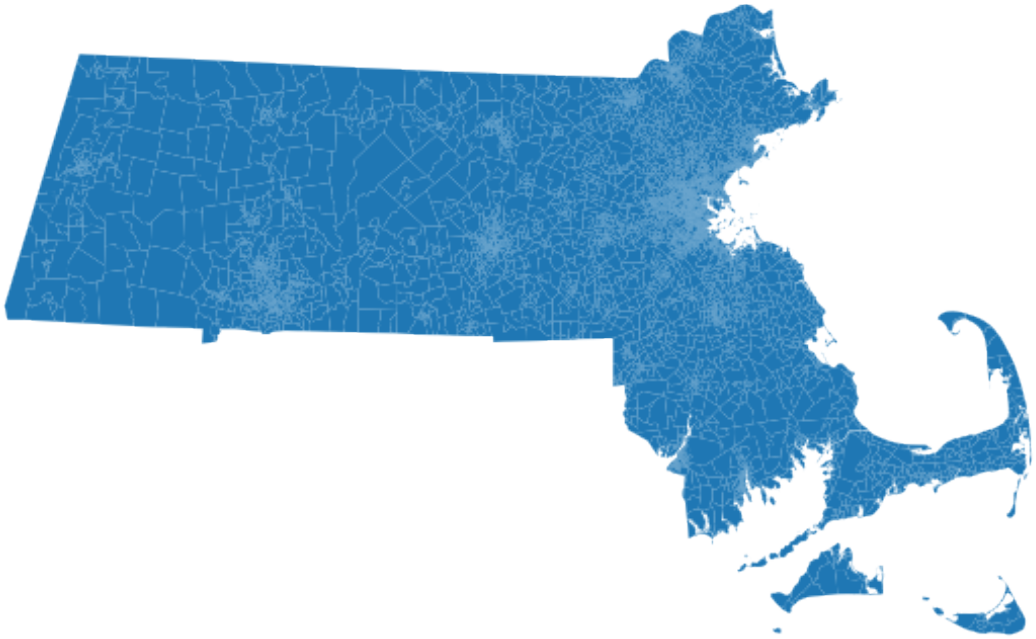
This graph was made from census block groups. We can use geopandas to open the zipped Shapefile.

```
>>> df = geopandas.read_file("zip://BG25.zip")
```

Inspecting the disconnected components

The next step in the process is to open the block group Shapefile in QGIS so we can see what's going on. For the purposes of this Jupyter notebook, we'll plot our GeoDataFrame `df`.

```
>>> import matplotlib.pyplot as plt
>>>
>>> # Change the projection so we can plot it:
>>> df.to_crs({"init": "epsg:26986"}, inplace=True)
>>>
>>> df.plot(figsize=(10, 10))
>>> plt.axis('off')
>>> plt.show()
```



png

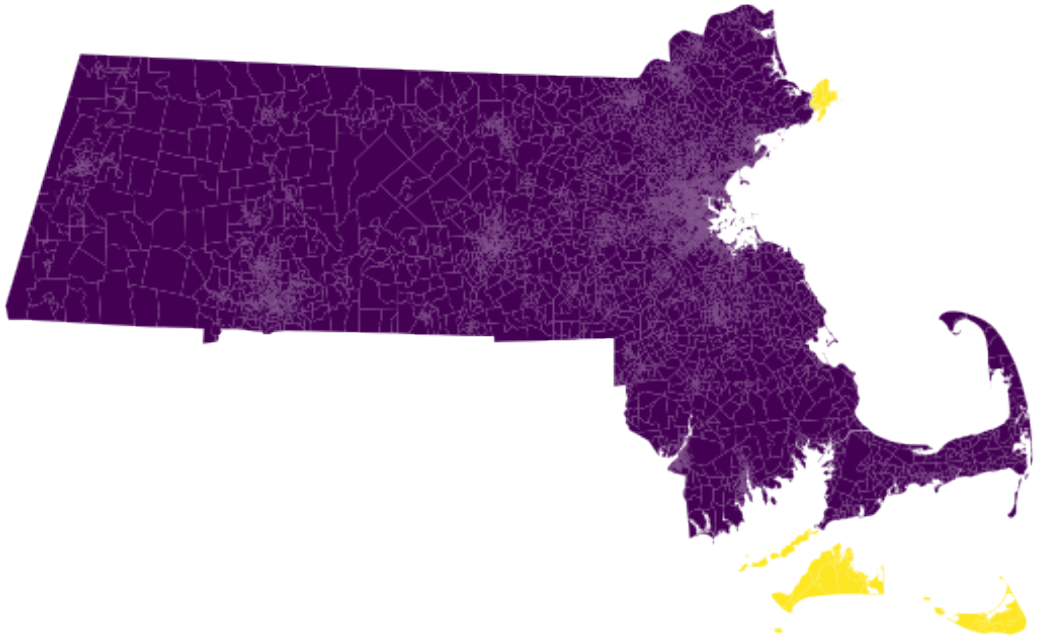
It's not immediately clear from this plot where the disconnected components are. But we can use our adjacency graph to get the GEOIDs of the disconnected nodes, which we can then highlight using QGIS (or geopandas).

Let's load our graph in again and verify that it's not connected.

```
>>> graph = Graph.from_json("./Block_Groups/BG25.json")
graph.py:216: UserWarning: Found islands (degree-0 nodes). Indices of islands: {2552,␣
↪3107}
    "Found islands (degree-0 nodes). Indices of islands: {}".format(islands)
>>> is_connected(graph)
False
```

Now we'll collect the GEOIDs of all of the nodes and use them to highlight the block groups that we need to connect.

```
>>> components = list(connected_components(graph))
>>> [len(c) for c in components]
[4922, 25, 19, 11, 1, 1]
>>> biggest_component_size = max(len(c) for c in components)
>>> problem_components = [c for c in components if len(c) != biggest_component_size]
>>> problem_nodes = [node for component in problem_components for node in component]
>>> problem_geoids = [graph.nodes[node]["GEOID10"] for node in problem_nodes]
>>>
>>> is_a_problem = df["GEOID10"].isin(problem_geoids)
>>> df.plot(column=is_a_problem, figsize=(10, 10))
>>> plt.axis('off')
>>> plt.show()
```



png

We can do the same thing in QGIS by using the "Select by Expression" feature with an expression like this:

```
"GEOID10" in (
 '250092215001',
 '250092216001',
 '250092214003',
 '250092213001',
 '250092214002',
 '250092215002', ...
)
```

where we've copied and pasted the GEOIDs from our `problem_geoids` variable into the `...` part between the parentheses.

Then click "Select features" and you should see these same features highlighted on the map.

Adding edges to the graph

Now comes the fun part! We want to identify some edges that we can add to make the graph connected. To do this, we want to:

- Zoom into the areas where these disconnected block groups (highlighted in yellow) are close to block groups in the main connected component (in purple)
- Use the "Identify features" tool to inspect the attributes of the yellow and purple block groups that we want to connect
- Copy the `"GEOID10"` attribute values for both of the units we want to connect

Once we have these GEOIDs, we can find the corresponding nodes in the graph and add that edge. For the Massachusetts example, here are two GEOIDs of nodes that I want to connect that I found in QGIS:

```
>>> purple_geoid = "250010149001"
>>> yellow_geoid = "250072004006"
```

Next, we find the corresponding node using this GEOID. Let's write a function to do this to save some mental space:

```
>>> def find_node_by_geoid(geoid, graph=graph):
>>>     for node in graph:
...         if graph.nodes[node]["GEOID10"] == geoid:
...             return node
```

```
>>> purple_node = find_node_by_geoid(purple_geoid)
>>> yellow_node = find_node_by_geoid(yellow_geoid)
>>> purple_node, yellow_node
(955, 2552)
```

And now we can finally add the edge `(955, 2552)` to connect these nodes:

```
>>> graph.add_edge(purple_node, yellow_node)
```

Let's use `connected_components` to see if we've made our graph more connected.

```
>>> [len(c) for c in connected_components(graph)]
[4923, 25, 19, 11, 1]
```

Yay! We've connected one of the islands that GerryChain warned us about. Now we'll repeat this process of adding edges until the graph only has one connected component.

Here are all the pairs of GEOIDs that I found that I wanted to connect:

```
>>> geoids_i_found = [
...     ("250072004001", "250072004006"),
...     ("250072003001", "250199503071"),
...     ("250092218002", "250092219022"),
...     ("250259901010", "250251803013")
... ]
```

Let's add them:

```
>>> edges_to_add = [(find_node_by_geoid(u), find_node_by_geoid(v)) for u, v in geoids_
↪i_found]
>>> for u, v in edges_to_add:
...     graph.add_edge(u, v)
```

And let's verify that the graph is connected:

```
>>> assert len(list(connected_components(graph))) == 1
>>> is_connected(graph)
True
```

Hooray! The graph is connected now!

Now we can run a proper Markov chain, without deleting any people from the graph.

Discontiguous plans

In addition to connectivity problems in the actual graph, you may also need to think about discontiguities in districting plans. That is, if we want to use a real-life plan as our initial state in GerryChain, we will want it to be contiguous, so

we need to make sure that our graph structure has the right edges in place for that to be true.

The process for fixing discontiguous plans is similar to the above process. The only difference is in how we identify the problematic nodes. GerryChain provides a function `contiguous_components` that takes a Partition and returns the contiguous components of each district.

Here's how we can find those components, using a random example plan with 2 districts for Massachusetts, just to see what a discontiguous plan looks like:

```
>>> from gerrychain.constraints.contiguity import contiguous_components, contiguous
>>> from gerrychain import Partition
>>> assignment = {}
>>> for node in graph:
...     if float(graph.nodes[node]["INTPTLAT10"]) < 42.356767:
...         assignment[node] = 0
...     else:
...         assignment[node] = 1
>>> discontiguous_plan = Partition(graph, assignment)
```

Now we'll verify that the plan does not pass our contiguity test, and examine the two contiguous components:

```
>>> contiguous(discontiguous_plan)
False
>>> contiguous_components(discontiguous_plan)
{0: [<Graph [2960 nodes, 7889 edges]>,
  <Graph [1 nodes, 0 edges]>,
  <Graph [1 nodes, 0 edges]>],
 1: [<Graph [2010 nodes, 5422 edges]>, <Graph [7 nodes, 15 edges]>]}
```

For any district with more than one contiguous component, you'll want to do the exact same process that we did with the overall graph above: add edges until there is only one contiguous component.

If the starting plan is not important to you, then you might want to use a function like `recursive_tree_part` to generate a starting plan from scratch.

1.2.7 Using `maup` to use a real-life plan in GerryChain

To generate an ensemble of districting plans using GerryChain, we need a starting point for our Markov chain. GerryChain gives you functions like `recursive_tree_part` to generate such plans from scratch, but you may want to use an actual districting plan from the real world instead. You also may want to compare a real-life plan to your ensemble.

`maup` is MGGG's package for doing common spatial data operations with redistricting data. We'll use this package to assign our basic geographic units to the districts in the plan we're interested in. Our steps are:

- Download a Shapefile of the districting plan we're interested in,
- Use `maup` to assign the our basic units to the districts in that plan, and then
- Create a GerryChain Partition using that assignment.

```
>>> import maup
>>> import geopandas
>>> import matplotlib.pyplot as plt
```

Downloading a plan

For this guide, we'll assign Massachusetts's block groups to the 2010 Massachusetts State Senate districting plan. We can use geopandas to download this Shapefile straight from the TIGER/Line files on the U.S. Census Bureau's website:

```
>>> districts = geopandas.read_file("https://www2.census.gov/geo/tiger/TIGER2012/SLDU/
↪tl_2012_25_sldu.zip")
```

Getting the assignment

Now we can use `maup` to assign our units to districts. See the `maup` documentation for another example and more information.

First we'll load our units as a GeoDataFrame:

```
>>> units = geopandas.read_file("zip://./BG25.zip")
```

Now we'll use `maup.assign` to get an assignment from units to districts:

```
>>> assignment = maup.assign(units, districts)
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-4-23d58953eccb> in <module>
----> 1 assignment = maup.assign(units, districts)


crs.py in wrapped(*args, **kwargs)
      9             raise TypeError(
     10                 "the source and target geometries must have the same CRS. {}
↪{}".format(
---> 11                     geoms1.crs, geoms2.crs
     12                 )
     13             )


TypeError: the source and target geometries must have the same CRS. {'proj': 'aea',
↪'lat_1': 29.5, 'lat_2': 45.5,'lat_0': 37.5, 'lon_0': -96, 'x_0': 0, 'y_0': 0, 'ellps
↪': 'GRS80', 'units': 'm', 'no_defs': True} {'init':'epsg:4269'}
```

Oh no! This error is telling us that the coordinates in our `units` and our `districts` are stored in different coordinate reference systems (different projections). We can fix this by setting the `units` CRS to match the `districts` CRS:

```
>>> units.to_crs(districts.crs, inplace=True)
```

Now let's try again:

```
>>> assignment = maup.assign(units, districts)
```

Yay! It worked!

Let's use the pandas `.isna()` method to see if we have any units that could not be assigned to districts:

```
>>> assignment.isna().sum()
0
```

This means that every unit was successfully assigned. If our basic units were too large to get a meaningful assignment, or if the districts did not cover all of our units (e.g. if our units included parts of the Atlantic Ocean but the districts did not), then we would have units with NA assignments that we would need to make decisions about.

We'll save the assignment in a column of our `units` GeoDataFrame:

```
>>> units["SENDIST"] = assignment
```

Creating a `Partition` with the real-life assignment

Now we are ready to use this assignment in GerryChain. We'll start by loading in our Graph from a pre-saved JSON file:

```
>>> from gerrychain import Graph, Partition
>>> graph = Graph.from_json("./Block_Groups/BG25.json")
```

In order to get this assignment onto the graph, we need to match the nodes of our graph to the geometries in `units` by their GEOIDs. We can use the `graph` object's `.join()` method to do this matching automatically:

```
>>> graph.join(units, columns=["SENDIST"], left_index="GEOID10", right_index="GEOID10
↪")
```

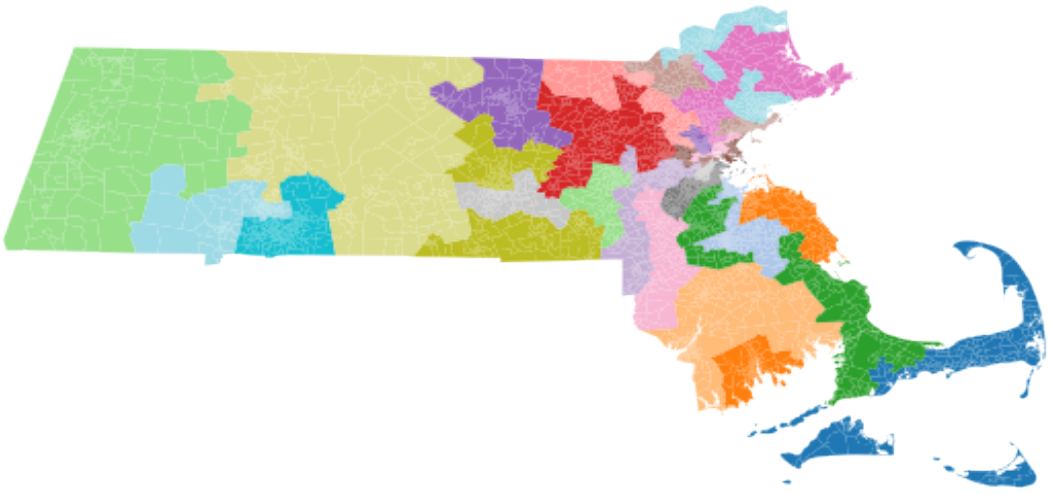
The `left_index` and `right_index` arguments tell the `.join()` method to use the `GEOID10` node attribute and the `GEOID10` column of `units` to match the records in `units` to the nodes of our graph.

Now `graph` has a node attribute called `SENDIST` that we can use to create a Partition:

```
>>> real_life_plan = Partition(graph, "SENDIST")
```

To check our work, let's plot the `real_life_plan` using the Partition's `.plot()` method:

```
>>> real_life_plan.plot(units, figsize=(10, 10), cmap="tab20")
>>> plt.axis('off')
>>> plt.show()
```



png

That looks like a districting plan! Woohoo!

Troubleshooting possible issues

If the plan looked like random noise or confetti, then we might suspect that something had gone wrong. The two places we would want to look for problems would be:

- the `graph.join` call, which would go wrong if the GEOIDs did not match up correctly, or
- the final `real_life_plan.plot` call, which would go wrong if the GeoDataFrame's index did not match the node IDs of our graph in the right way.

We could inspect both issues by making sure that the records with matching IDs actually referred to the same block groups.

You also might run into problems when you go to run a Markov chain using the partition we made. If the districts are not contiguous with respect to your underlying graph, you would want to add edges (within reason) to make the graph agree with the notion of contiguity that the real-life plan uses. See What to do about islands and connectivity for a guide to handling those types of issues.

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

1.2.8 API Reference

Table of Contents

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=False*, *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'*, *cols_to_add=None*, *reproject=False*, *ignore_errors=False*)

Creates the adjacency *Graph* of geometries described by *dataframe*. The areas of the polygons are in-

cluded 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"
- **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.
- **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.

Parameters

- **json_file** – Path to target JSON file.

- **include_geometry_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.

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

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.

Variables

- **graph** (*gerrychain.Graph*) – The underlying graph.
- **assignment** (*gerrychain.Assignment*) – Maps node IDs to district IDs.
- **parts** (*dict*) – Maps district IDs to the set of nodes in that district.
- **subgraphs** (*dict*) – Maps district IDs to the induced subgraph of that district.

Parameters

- **graph** – Underlying graph.
- **assignment** – Dictionary assigning nodes to districts.
- **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

plot(*geometries=None*, ***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.

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.
- **assignment** – Dictionary assigning nodes to districts.
- **updaters** – Dictionary of functions to track data about the partition. The keys are stored as attributes on the partition class, which the functions compute.

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.

Proposals

gerrychain.proposals.**recom**(*partition*, *pop_col*, *pop_target*, *epsilon*, *node_repeats=1*, *method=<function bipartition_tree>*)

ReCom proposal.

Description from MGGG's 2018 Virginia House of Delegates report: At each step, we (uniformly) randomly select a pair of adjacent districts and merge all of their blocks in to a single unit. Then, we generate a spanning tree for the blocks of the merged unit with the Kruskal/Karger algorithm. Finally, we cut an edge of the tree at random, checking that this separates the region into two new districts that are population balanced.

Example usage:

```
from functools import partial
from gerrychain import MarkovChain
from gerrychain.proposals import recom

# ...define constraints, accept, partition, total_steps here...

# Ideal population:
pop_target = sum(partition["population"].values()) / len(partition)

proposal = partial(
    recom, pop_col="POP10", pop_target=pop_target, epsilon=.05, node_repeats=10
)

chain = MarkovChain(proposal, constraints, accept, partition, total_steps)
```

gerrychain.proposals.**spectral_recom**(*partition*, *weight_type=None*, *lap_type='normalized'*)

Spectral ReCom proposal.

Uses spectral clustering to bipartition a subgraph of the original graph formed by merging the nodes corresponding to two adjacent districts.

Example usage:

```
from functools import partial
from gerrychain import MarkovChain
from gerrychain.proposals import recom

# ...define constraints, accept, partition, total_steps here...


proposal = partial(
    spectral_recom, weight_type=None, lap_type="normalized"
)

chain = MarkovChain(proposal, constraints, accept, partition, total_steps)
```

gerrychain.proposals.**propose_chunk_flip**(*partition*)

Chooses a random boundary node and proposes to flip it and all of its neighbors

Parameters **partition** – The current partition to propose a flip from.

Returns a proposed next ~*gerrychain.Partition*

gerrychain.proposals.**propose_random_flip**(*partition*)

Proposes a random boundary flip from the partition.

Parameters **partition** – The current partition to propose a flip from.

Returns a proposed next ~*gerrychain.Partition*

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
UpperBound	Upper bounds on numeric constraints
LowerBound	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)
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.**LowerBound**(*func*, *bound*)

Wrapper for numeric-validators to enforce lower limits.

This class is meant to be called as a function after instantiation; its return is `True` if the numeric validator is within a set lower limit, and `False` otherwise.

Parameters

- **func** – Numeric validator function. Should return a comparable value.
- **bounds** – Comparable lower bound.

class gerrychain.constraints.**SelfConfiguringLowerBound**(*func*, *epsilon=0.05*)

Wrapper for numeric-validators to enforce automatic lower limits.

When instantiated, the initial lower bound is set as the initial value of the numeric-validator minus some configurable ϵ.

This class is meant to be called as a function after instantiation; its return is `True` if the numeric validator is within a set lower limit, and `False` otherwise.

Parameters

- **func** – Numeric validator function.
- **epsilon** – Initial "wiggle room" that the validator allows.

class gerrychain.constraints.**SelfConfiguringUpperBound**(*func*)
Wrapper for numeric-validators to enforce automatic upper limits.

When instantiated, the initial upper bound is set as the initial value of the numeric-validator.

This class is meant to be called as a function after instantiation; its return is `True` if the numeric validator is within a set upper limit, and `False` otherwise.

Parameters **func** – Numeric validator function.

class gerrychain.constraints.**UpperBound**(*func*, *bound*)
Wrapper for numeric-validators to enforce upper limits.

This class is meant to be called as a function after instantiation; its return is `True` if the numeric validator is within a set upper limit, and `False` otherwise.

Parameters

- **func** – Numeric validator function. Should return a comparable value.
- **bounds** – Comparable upper bound.

gerrychain.constraints.**contiguous**(*partition*)
Check if the parts of a partition are connected using `networkx.is_connected()`.

Parameters **partition** – The proposed next *Partition*

Returns whether the partition is contiguous

Return type bool

gerrychain.constraints.**contiguous_bfs**(*partition*)
Checks that a given partition's parts are connected as graphs using a simple breadth-first search.

Parameters **partition** – Instance of Partition

Returns Whether the parts of this partition are connected

Return type bool

gerrychain.constraints.**single_flip_contiguous**(*partition*)
Check if swapping the given node from its old assignment disconnects the old assignment class.

Parameters **partition** – The proposed next *Partition*

Returns whether the partition is contiguous

Return type bool

We assume that *removed_node* belonged to an assignment class that formed a connected subgraph. To see if its removal left the subgraph connected, we check that the neighbors of the removed node are still connected through the changed graph.

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.

gerrychain.constraints.**districts_within_tolerance**(*partition*, *attribute_name='population'*, *percentage=0.1*)

Check if all districts are within a certain percentage of the "smallest" district, as defined by the given attribute.

Parameters

- **partition** – partition class instance
- **attrName** – string that is the name of an updater in partition
- **percentage** – what percent difference is allowed

Returns whether the districts are within specified tolerance

Return type bool

gerrychain.constraints.**no_vanishing_districts**(*partition*)

Require that no districts be completely consumed.

gerrychain.constraints.**refuse_new_splits**(*partition_county_field*)

Refuse all proposals that split a county that was previous unsplit.

Parameters **partition_county_field** – Name of field for county information generated by *county_splits()*.

gerrychain.constraints.**within_percent_of_ideal_population**(*initial_partition*, *percent=0.01*, *pop_key='population'*)

Require that all districts are within a certain percent of "ideal" (i.e., uniform) population.

Ideal population is defined as "total population / number of districts."

Parameters

- **initial_partition** – Starting partition from which to compute district information.
- **percent** – (optional) Allowed percentage deviation. Default is 1%.
- **pop_key** – (optional) The name of the population *Tally*. Default is `"population"`.

Returns A *Bounds* constraint on the population attribute identified by `pop_key`.

class gerrychain.constraints.**Bounds**(*func*, *bounds*)

Wrapper for numeric-validators to enforce upper and lower limits.

This class is meant to be called as a function after instantiation; its return is `True` if the numeric validator is within set limits, and `False` otherwise.

Parameters

- **func** – Numeric validator function. Should return an iterable of values.
- **bounds** – Tuple of (lower, upper) numeric bounds.

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.

class gerrychain.constraints.**UpperBound**(*func*, *bound*)

Wrapper for numeric-validators to enforce upper limits.

This class is meant to be called as a function after instantiation; its return is `True` if the numeric validator is within a set upper limit, and `False` otherwise.

Parameters

- **func** – Numeric validator function. Should return a comparable value.
- **bounds** – Comparable upper bound.

class gerrychain.constraints.**LowerBound**(*func*, *bound*)

Wrapper for numeric-validators to enforce lower limits.

This class is meant to be called as a function after instantiation; its return is `True` if the numeric validator is within a set lower limit, and `False` otherwise.

Parameters

- **func** – Numeric validator function. Should return a comparable value.
- **bounds** – Comparable lower bound.

class gerrychain.constraints.**SelfConfiguringLowerBound**(*func*, *epsilon=0.05*)

Wrapper for numeric-validators to enforce automatic lower limits.

When instantiated, the initial lower bound is set as the initial value of the numeric-validator minus some configurable ϵ.

This class is meant to be called as a function after instantiation; its return is `True` if the numeric validator is within a set lower limit, and `False` otherwise.

Parameters

- **func** – Numeric validator function.
- **epsilon** – Initial "wiggle room" that the validator allows.

class gerrychain.constraints.**SelfConfiguringUpperBound**(*func*)

Wrapper for numeric-validators to enforce automatic upper limits.

When instantiated, the initial upper bound is set as the initial value of the numeric-validator.

This class is meant to be called as a function after instantiation; its return is `True` if the numeric validator is within a set upper limit, and `False` otherwise.

Parameters **func** – Numeric validator function.

class gerrychain.constraints.**WithinPercentRangeOfBounds**(*func*, *percent*)

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(
```

(continues on next page)

(continued from previous page)

```
    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.

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
```

(continues on next page)

(continued from previous page)

```
# 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

efficiency_gap()

Computes the efficiency gap for this ElectionResults object.

mean_median()

Computes the mean-median score for this ElectionResults object.

mean_thirdian()

Computes the mean-thirdian score for this ElectionResults object.

partisan_bias()

Computes the partisan bias for this ElectionResults object.

partisan_gini()

Computes the Gini score for this ElectionResults object.

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.

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.

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.

class gerrychain.tree.**Cut**(*edge*, *subset*)
Create new instance of Cut(edge, subset)

edge
Alias for field number 0

subset
Alias for field number 1

gerrychain.tree.**bipartition_tree**(*graph*, *pop_col*, *pop_target*, *epsilon*, *node_repeats=1*, *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.**bipartition_tree_random**(*graph*, *pop_col*, *pop_target*, *epsilon*, *node_repeats=1*, *spanning_tree=None*, *choice=<bound method Random.choice of <random.Random object>>*)
This is like *bipartition_tree()* except it chooses a random balanced cut, rather than the first cut it finds.

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=1*, *method=<function bipartition_tree>*)

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

Metrics

gerrychain.metrics.**mean_median**(*election_results*)

Computes the Mean-Median score for the given ElectionResults. A positive value indicates an advantage for the first party listed in the Election's parties_to_columns dictionary.

gerrychain.metrics.**partisan_bias**(*election_results*)

Computes the partisan bias for the given ElectionResults. The partisan bias is defined as the number of districts with above-mean vote share by the first party divided by the total number of districts, minus 1/2.

gerrychain.metrics.**partisan_gini**(*election_results*)

Computes the partisan Gini score for the given ElectionResults. The partisan Gini score is defined as the area between the seats-votes curve and its reflection about (.5, .5).

gerrychain.metrics.**efficiency_gap**(*results*)

Computes the efficiency gap for the given ElectionResults. A positive value indicates an advantage for the first party listed in the Election's parties_to_columns dictionary.

gerrychain.metrics.**polsby_popper**(*partition*)

Computes Polsby-Popper compactness scores for each district in the partition.

gerrychain.metrics.**wasted_votes**(*party1_votes*, *party2_votes*)
Computes the wasted votes for each party in the given race. :party1_votes: the number of votes party1 received in the race :party2_votes: the number of votes party2 received in the race

1.2.9 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.

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.

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.

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...
```

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

M

N

P

R

S

T

U

V

W