GerryChain Practice Day 2: Conditionals and Constraints

Daryl DeFord

June 26, 2019

1 Introduction

The main purpose of today's session is to get some more experience working with the `Partition` class in GerryChain and the `constraints` that are used to define the state space of permissible partitions. This is also a good opportunity to practice using conditional expressions in python. This is a crucial step in building a good Markov chain sampler, since there are vastly more "bad" partitions than nice ones and keeping the bad ones out of our ensemble saves a lot of wasted computational effort.

One of the overall goals of these sessions is preparing you for the Your State project. Hopefully some of the functions you design in these sessions will make it in to your presentation ☺

1.1 Python Code

The initial code for these investigations is in a separate repo in the VRDI organization. You should be able to do everything in a Jupyter Notebook or whichever other IDE you used on Friday for the original templates. The more detailed guide is a good reference for more examples.

2 Proposals

The `proposal` method lies at the heart of the discrete MCMC approach to generating districting plans. As with the updaters and constraints, each proposal is a function that takes as input a `Partition` object[1] and returns a dictionary that maps node ids to their new district assignments. Only those nodes that change assignments need to be added to the dictionary. You can see the prebuilt collection of proposal functions to get a sense of the syntax here. You can read more theoretical discussion in Section 8.4 of this and more practical discussion in Section 6 of this.

The template provided in the Day3 has a couple examples of bad proposals from the GerryChain Guide. Note that we are using grids for the template today because it makes it easier to visualize what is happening at each step - when we look at geographic partitions, the interesting information is often hidden in the small nodes in the urban areas. The basic setup for many of these proposals is similar: select a set of nodes to flip and then choose from among their neighbors. As we saw in the original Grid Templates, one of the things we are most interested in is comparing the properties of various proposals. Below are some ideas to compare the examples in the current template, as well as to ReCom. Note that you may want to loosen the population constraint to see more exciting behavior.

- How much time does it take to run each chain?
- Which chain outputs better looking partitions?
- Which chain sees more seats values with the randomized vote data?
- What are the average boundary lengths of the final plans returned by the chains?
- If you use a stronger compactness constraint, which proposal rejects more steps?

[1] Are you starting to see a pattern here?

3 Smarter Flips

The other proposal distribution included in the template is designed in such a way that it generates a reversible Markov chain. Notice that it uses an updater to keep track of the (node, district) pairs that at least have the potential to lead to contiguous partitions. Run some chains with this proposal and record some partisan metrics for the Pink-Purple and Green-Yellow elections. For each metric, compute the proportion of observed values that are less than the initial value. This procedure replicates the p-value test introduced in this paper.

One interesting idea is to try to use the proposal process to optimize the partition for population balance or compactness. This is likely to be particularly relevant in states with large state legislatures, as building seed plans with large numbers of districts can be difficult, even on precinct-sized units. One possible approach is to generate an unbalanced plan and then try to use

- Make a proposal that tries to improve population balance. Use the recursive tree method to generate a seed with 10% population imbalance and then see if your proposal actually improves the plans in the ensemble.
- Make a proposal that tries to improve compactness. Run the uniform proposal for 10000 steps and then switch to your new proposal. Plot the number of cut edges as the chains run.

Another way to extend the flip method is to try to perform several adjacent flips at once. This doesn't appear to solve all of the problems with flip but it does at least provide some hope for understanding the best case scenario for the flip walk[2]

- Make a proposal that tries to flip all the nodes that lie along a single boundary between two districts. Note that the mediocre proposal tries one version of this - you should try to move all of the nodes the same "direction."
- Make a proposal that tries to flip all of the nodes that belong to the boundary of a single district. Again, you should try to have all of the nodes move the same direction.
- Make a proposal that tries to flip all of the cut edges in the plan (or at least adds each boundary node to the flips dictionary). This is likely to get stuck pretty quickly unless you implement it cleverly[3].

4 ReCom Proposals

One of the main advantages of ReCom is that it changes the assignments of many nodes at once and completely obliterates the original boundary between the two nodes. Although we have mostly used the spanning tree based version of ReCom there are many possible ways to do the bipartitioning. Some examples are here and there is a spectral method implemented in your installation of GerryChain. Additionally, there are many possibilities for choosing which districts or pairs of districts to merge. Here are some suggestions to

- Write a proposal that merges three districts at once and then uses the recursive tree method to split them at each step.
- Write a proposal that forms a maximal matching on the dual graph of the districts and pairs all of the endpoints of the matching at once.
- Generate a ReCom proposal that uses the most population balanced vertical or horizontal line to split the merged districts.
- Modify the spectral ReCom proposal to put arbitrary weights on the edges of the subgrpah corresponding to the merged districts.

[2] which isn't very good.
[3] in which case it will get stuck slightly less quickly