Recombination Knapsack Experimental Report

Eric Wang

August 16, 2024

Contents

1 Introduction

This report delves into the complex challenge of designing voting districts, an essential component of fair representation in democratic systems. In many areas around the U.S., voting districts are composed of smaller demographic areas, and are often required to adhere to various constraints on population. For example, Chicago's 50 wards are constructed out of over 2,000 precincts, and are each required to have populations that deviate no more than 5% from the ideal[1]. In order to investigate the feasibility of this restriction under different conditions, we present a tool to experimentally test districting plans using Monte-Carlo Markov Chain (MCMC) sampling through the `recomb_knapsack` Python package detailed in this report.

We approach this challenge through the lens of a new optimization problem we will call a "partitioning knapsack" problem. By framing the division of voting districts within this mathematical context, we aim to uncover new strategies for maintaining close-to-equal populations across districts. By viewing voting districts as partitions of smaller geographic components, we gain a deeper understanding of how to achieve fair and effective political boundaries. This approach not only enhances our theoretical grasp of districting challenges but also offers practical tools for improving the fairness and accuracy of the redistricting process.

1.1 Partitioning Knapsack Problem

In the traditional knapsack problem over a set of items, every item is associated with a weight, and a "solution" simply contains a subset of those items such that the total weight of the items does not exceed a predefined capacity of the knapsack. Common extensions to this problem include associating every item with some value to maximize, or allowing items to be selected multiple times. However for this context, we will define a **"partitioning knapsack"** problem, where rather than considering a single knapsack, we instead consider multiple knapsacks that will partition the entire set of items such that each knapsack contains (either approximately or exactly) an equal fraction of the total over all items. More formally, we define it as follows:

- Let X be the set of n items with weights $x_1, x_2, ..., x_n$, the items to consider in the problem.
- Let k be the number of knapsacks the problem seeks to generate.
- Optionally define a *tol* value to represent the allowed tolerance for each knapsack partition. The ideal total weight of each partition is $\frac{\sum X}{k}$, and so the allowed partition weight (or the **tolerable range**) will be $\frac{\sum X}{k} \pm tol$.

In simple terms, the partitioning knapsack problem over a set of items X is parameterized by the number of knapsacks k and the allowed tolerance *tol*, and a valid solution consists of

[1] See `https://mggg.org/chicago` for more exploration into this topic.

the items of X divided into k partitions (every item is used exactly once), where the total weight of each of the k partitions is within the tolerable range $\frac{\sum X}{k} \pm tol$.

1.2 Terminology

For this write-up, we will define a few key terms and language to describe the "partitioning-knapsack" problem and relate it to the real-world application of partitioning U.S. census tracts to voting districts:

- Each individual item in the set X will be known as **precincts**, with the weights $x_1, ..., x_n$ representing the population count of each precinct.
- A **proposed solution** to the partitioning knapsack problem contains k sets of items in X, where k is a number defined in the problem parameters. Each set of precincts is known as a **district**, and precinct in X must appear in exactly one district[2].
- A **valid district** containing items X_i must have a sum of weights $\sum X_i$ that fall within the interval $[\frac{\sum X}{k} - tol, \frac{\sum X}{k} + tol]$. Naturally, a **valid solution** is a proposed solution where all k districts are valid.

For the purposes of the following experiments, we are primarily concerned simply with whether a proposed solution is valid or not, and properties that affect how many valid solutions exist for a certain problem configuration. Other questions, such as developing an algorithm to find a valid solution, is certainly worth investigating, though not the focus of this write-up.

2 Repository Demo

The current repository can be located at the `recomb_knapsack` GitHub repository, containing the code developed over the duration of the project. The functional portions are mainly contained within two subdirectories `recomb_knapsack/utils` and `recomb_knapsack/sampling`. As the names suggest, `recomb_knapsack/utils` contains a number of useful functions for experimentation with this subject area, and `recomb_knapsack/sampling` contains the core recombination code for MCMC sampling over knapsack solutions.

2.1 'utils' Directory

This directory contains a few of the key functions used to aid in experiments. Most notable are the `generate_epsilon()` and `generate_gamma()` functions, which are ways to generate data according to the two data distributions that this experiment investigates.

[2] "Precincts" to "districts" is just one example of an application for partitioning knapsack. This concept can easily be applied to blocks, block groups, wards, etc.

- `generate_epsilon()`: This function samples from the `Normal(1, ep)` distribution, where `ep` is passed as a parameter. This distribution is used to generate examples where precincts are all similarly sized (close to 1).

- `generate_gamma()`: This function samples from a gamma distribution with shape parameter `k=2.34`, scale parameter θ=`814.8`, and shifted by `-200.21`. This configuration creates a distribution that closely matches the national precinct population from the 2020 U.S. Census, as reported by Prof. Daryl Deford[3].

  As seen below, the distribution of U.S. precinct populations can be closely mapped to a gamma or log normal distribution. For the purposes of this experiment, we decide to only investigate the gamma distribution.

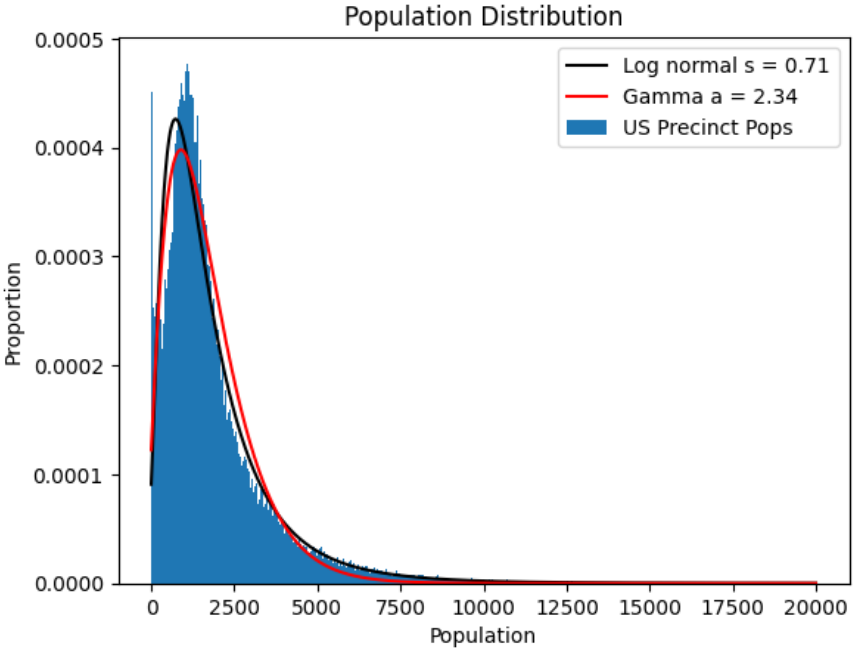

Figure 1: Mapping a gamma distribution to U.S. precinct sizes

- `generate_gamma_clean()`: This function draws from the same distribution as `generate_gamma()`, however all negative data points are set to `0`, and all data points are rounded to the nearest integer. This simulates real population counts while still closely following the aforementioned gamma distribution.

2.2 'sampling' Directory

This directory contains the functions used to perform the MCMC sampling as part of the experiment. We use a recombination strategy with MCMC sampling, where every state is represented by a proposed solution (partitioning) of precincts to districts, and we examine

[3]https://github.com/drdeford

the frequency of valid solutions to determine how "easy" it is to find a valid solution given certain problem parameters. In general, the methodology for a run of MCMC sampling goes as follows, where n is the number of precincts and k is the number of districts.

1. Randomly assign precincts to districts to create a proposed solution that acts as your starting state. This is done by simply taking the precincts in a random order and assigning $\frac{n}{k}$ precincts to each district.
2. At each iteration, perform a recombination step by selecting two districts in the current proposed solution, shuffling them, then creating two new districts using only the precincts from the original districts.
3. If this results in a valid solution, record the current partitions and state. Repeat step 2 for as many iterations as desired.

In this process, step 2 presents an interesting point of investigation, as how you perform the recombination step may have a large impact on the frequency and nature of valid solutions. This repository supports two types of recombination steps, `first_valid` and `equal_weights`, detailed below:

- `first_valid`: After shuffling precincts, process precincts one at a time and greedily accept them into `res1`. Terminate the process as soon as `res1` becomes a valid district, and assign all remaining precincts into `res2`. This step has a chance to fail, in which case the districts will remain unchanged.
- `equal_weights`: After shuffling precincts, process precincts one at a time and greedily accept them into `res1`. Terminate the process as soon as the sum of precincts in `res1` exceeds half the total weight of all precincts, and assign all remaining precincts into `res2`. Unlike `first_valid`, this step cannot fail, but it does not guarantee any valid districts.

Both of these methods are greedy in order to reduce the computational cost of the recombination step, but there are also many potential non-greedy recombination methods that could perform very differently.

3 Future Work

Future work can look to delve deeper into exploring different recombination strategies when sampling, potentially finding strategies that offer different outcomes or more efficient analysis. Additionally, more exploration into the space of possible solutions, in particular the range of variance between valid solutions, may provide further insight into what characteristics of the input data lead to diverse solutions.