A k-medoids Approach to Exploring Districting Plans

Jared Grove, Suely P. Oliveira, Anthony Pizzimenti, and David E. Stewart

Abstract

As the fight against unconstitutional partisan gerrymandering mounts, researchers and legislators alike continue the search for methods of drawing fair districting plans. A districting plan is a partition of a state's subdivisions (e.g. counties, voting precincts, etc.). By modeling these districting plans as graphs, they are easier to create, store, and operate on. In this paper, we present a variant on the k-medoids algorithm where, given a set of initial medoids, we find an optimal partition of the graph's vertices, admitting a districting plan.

Contents

1 Introduction

1.1 Abbreviated History

The practice of gerrymandering is not a new one. With a rich history in the political arena, gerrymandering can wear many faces: prison, partisan, racial, and incumbent gerrymandering are all practicable and, oftentimes, legal. Generally used as a tactic for political parties to retain legislative power, it stands in stark contrast to traditional ideals of representative government; gerrymandering allows politicians to pick their voters rather than voters pick their politicians. In light of increasing attention paid to this issue and the socio-political impact it exerts, it is pertinent to explore ways of detecting, preventing, and remedying gerrymandered voting maps. In this paper, we present a method of generating potential districting plans by an adjusted k-medoids algorithm.

1.2 Introductory Concepts

At its most basic, a districting plan is a way of chopping up a state into large chunks, each of which is made up of a number of state subdivisions (e.g. counties, voting precincts, Voter Tabulation Districts, etc.). In this paper, we will be dealing mainly with abstract states, so it is necessary to outline some familiar concepts relevant to modeling this problem.

Suppose we have a state S with n subdivisions such that $S = \{s_1, \ldots, s_n\}$, and we want to divide S into k districts. We can introduce a labeling function $l : S \to \mathbb{N}$, where l maps each subdivision to a label $\{1, \ldots, k\}$. This function admits a partition $\mathcal{D}$ of S; this partition is a districting plan.

Now suppose we want to perform computations on and evaluate metrics of different districting plans. In order to do so, we use S's dual graph G_S – that is, the graph whose vertices are the subdivisions of S's map – to reflect the geographical and mathematical relationships present in S. Further, we can apply the same labeling function l to the vertices $\{s_1, \ldots, s_n\}$ of G_S such that the same partition $\mathcal{D}$ is present; this, in turn, partitions G_S into k nontrivial connected components representing S's districts.

1.3 Contemporary Methods

The transition from abstract states to real-world data can be quite daunting. New York State, for example, has more than 11,000 Voter Tabulation Districts (VTDs), which are the subdivisions used to compose New York's Congressional districts. Attempting to enumerate the set of districting plans for a given state is not practicable (especially with limited time and computing resources); in light of this, finding a method to explore a state's possible districting plans is essential.

By using a Monte-Carlo Markov Chain walk method, it is possible to explore a number of districting plans with reasonable efficiency. Taking the graph G_S, we can impose an initial districting plan $\mathcal{D}_0$. Then, at each step of the Markov chain, randomly pick a vertex $s_i \in S$ such that s_i is on the boundary between two districts; i.e there exists an edge (s_i, s_j) such that $l(s_i) \neq l(s_j)$. Then, we "flip" s_i to the same district as s_j such that $l(s_i) = l(s_j)$. We continue in this manner until the desired distribution of districting plans has been explored. `Gerrychain`, software to perform this exploration on a given state, provides users the ability to supply their own state-level data (e.g.

vote distributions, demographics, geographical boundaries, etc.) and custom functions to collect metrics on the districting plans at each step of the chain. This method, however, requires an initial districting plan $\mathcal{D}_0$ to seed the MCMC walk.

1.4 k-medoids

Much like the well-known k-means method of data clustering, the k-medoids algorithm seeks to partition a set of n points into k clusters such that, for each cluster of points, the sum of squared distances between each point and the average of all points c (the *centroid*) in that cluster is minimized. k-medoids takes a similar approach, but instead of picking a centroid, a *medoid* is picked — a point m such that the some of squared distances between points in the cluster and m is minimized, where m is a point in the dataset.

2 Motivation for a k-medioids approach

3 Algorithms

In order use a k-medoids approach we had to implement it first. The pseudo code for the algorithms we used are as follows.

Data: Tree, Number of Medoids, Stopping Criteria
Result: Clusters for each of the best found Medioids
Select initial medoids;
Assign nodes to nearest medoid;
while not converged **do**
  Reassign_medoids[tree, clusters, medoids];
  Assign nodes to closest medoid;
end

Algorithm 1: KMedoids

The input to this algorithm is a tree, the desired number of medoids, and a stopping criteria. The stopping criteria we use is the number of times equivalent clusters appear in a row, this is a user defined number, but we choose two. To begin we randomly assign initial starting medoids and assign each node to the medoid that is closest. We achieve this by implementing a Breadth First Search (BFS) from each medoid simulaneously. Start at the first medoid and find all nodes within 1 step, then move the second medoid and find all nodes within 1 step, repeat for all medoids. Next, for the first medoid, find all nodes within 2-steps. Repeat for the rest of the medoids. Continue this process until all nodes have been assigned to a medoid. Once all the nodes have been assigned to a medoid, we have a set of clusters. The next step is to select new medoids. We implemented two different methods to achieve this, and will be comparing the two methods through the rest of the paper.

Data: Tree, Clusters, Medoids
Result: New medoids
for cluster in Clusters **do**
    Compute Diameter Path;
    Find midpoint of Diameter Path;
    Set midpoint as new Medoid;
end

Algorithm 2: Reassign_medoids- Method 1

The first method, which will be refered to as Method 1, uses the Diameter Path to select the new medoids. This is achieved by selecting a random node in a cluster, performing a Depth First Search (DFS) within the cluster, and selecting one of the nodes that is as far down the tree as possible. Then, perform a second intra-cluster DFS on that node. The most distant node from the DFS search will give the length of the Diameter Path. From there we compute the Diameter Path and select the node that is at the halfway point, or floor of the halfway point, on the Diameter Path as the new medoid for that cluster. Repeat this for all clusters to get a set of new medoids. Reassign clusters with the simultaneous BFS and repeat until equivalent sets of medoids and clusters appear a user specified number of times in a row.

Data: Tree, Clusters, Medoids
Result: New medoids
for cluster in Clusters **do**
    Compute distance between all nodes in cluster;
    Set new medoid to node with smallest total distance;
end

Algorithm 3: Reassign_medoids- Method 2

The second method we implemented, Method 2, works very differently. To select the new medoid it starts with the medoid of the cluster. Next select a neighbor of the medoid and cut the tree on the edge between the medoid and that neighbor. Count how many nodes are on the two resulting subtrees. The number of nodes on the subtree with the neighbor will be denoted as (medoid, neighbor) and the number of nodes on the side of the medoid will be denoted as (neighbor, medoid). Restore the cut edge and repeat for all neighbors of the medoid. Compute (medoid, neighbor)-(neighbor, medoid) for all neighbors. If this quantity is negative for all neighbors, the medoid will not change. If this quantity is positive for at least one nieghbor, move the medoid to the neighbor that maximizes that quantity. Repeat this process with the new medoid until no improvement can be found. The last node considered as a medoid will be returned.

4 Implementation

5 Tests and Results

To test the performance of the two methods we ran them on randomly generated trees from the python networkx library. We tested on trees of size 100, 500, $1,000$, $5,000$, and $10,000$ using numbers of medoids from the set $\{5, 10, 15, 20, 25\}$. The two methods ended up producing clusters that are equivalent. Thus, we will first focus on comparing the speed of the two methods here. The columns in Table 1 represent how many medoids, the number of nodes, the number of experiments, the maximum run time for Method 1, the average run time for Method 1, the maximum run time for Method 2, and the average run time for Method 2 respectively.

k	nodes	Number	MaxTime1	AvgTime1	MaxTime2	AvgTime2
5	100	50	0.06	0.02	0.007	0.003
5	500	50	0.504	0.11	0.07	0.015
5	1000	45	0.699	0.238	0.084	0.03
5	5000	25	12.879	3.862	0.329	0.131
5	10000	25	58.987	14.53	0.353	0.228
10	100	50	13.317	0.668	8.391	0.333
10	500	50	74.265	2.077	39.343	0.802
10	1000	45	2.976	0.646	0.092	0.029
10	5000	25	43.643	9.712	0.218	0.127
10	10000	25	382.557	48.074	0.368	0.232
15	100	50	14.138	0.734	9.161	0.187
15	500	50	10.996	1.232	0.042	0.015
15	1000	45	185.27	9.357	88.526	1.993
15	5000	25	46.3	13.591	0.184	0.114
15	10000	25	541.158	109.944	0.399	0.267
20	100	13	541.158	80.255	10.088	0.947
20	500	5	16.831	5.995	0.012	0.011
20	1000	5	67.274	16.056	0.023	0.023
20	5000	5	64.007	20.084	0.12	0.117
20	10000	5	530.554	314.034	0.313	0.261
25	100	5	15.225	3.499	11.27	2.256
25	500	5	84.481	50.058	0.021	0.015
25	1000	5	116.085	47.008	0.027	0.025
25	5000	5	1534.229	522.327	0.211	0.145
25	10000	5	2276.289	1030.422	0.342	0.296

Table 1: Method 1 vs Method 2

As we can see here, Method 2 vastly outspeeds Method 1 in nearly all cases. Method 2 has the fastest average run time in each of the scenarios we considered and has the lowest maximum run time as well. Furthermore, we can see that Method 2, on average, performs very well regardless of the number of medoids or nodes as the average run time is always under 2 seconds and is under 1 second in all but one case. On the other hand, Method 1 has much higher runtimes. This method seems to struggle when dealing with large number of nodes, but performs similarly to Method 2 when the number of nodes is small. Since recieving these results we have noticed that Method 1 has a tendency to come back to old solutions. In order to rectify this we are planning to implement some form of a Tabu search to remove this issue.

Next we will consider the clusters that the methods produced. The columns of the Table 2 represent the number of medoids, the number of nodes, the number of experiments, the average maximum intracluster distance, the average minimum intracluster distance, the average mean intracluster distance, the average maximum number of nodes in a cluster, and the average minimum number of nodes in a cluster respectively.

k	nodes	Number	MaxDist	MinDist	MeanDist	MaxSize	MinSize
5	100	50	16.94	2.22	7.74	36.46	7.54
5	500	50	29.98	3.1	12.9	187.54	37.42
5	1000	45	48.02	3.18	17.91	371.78	81.8
5	5000	25	59	2.92	21.12	1991.48	315.4
5	10000	25	53.04	2.6	19.56	3988.04	754.56
10	100	50	11.22	0.74	3.74	23.14	2.82
10	500	50	33.82	1.84	10.84	119.68	13.06
10	1000	45	45.47	1.89	14.29	260.16	18.4
10	5000	25	69.36	1.68	17.8	1241.16	113.64
10	10000	25	66.84	1.92	17.64	2164.92	267.36
15	100	50	9	0.5	2.18	16.7	1.74
15	500	50	30.76	1.12	9.08	80.14	7.24
15	1000	45	47.82	1.49	12.29	181.73	14.07
15	5000	25	56.8	1.32	15.84	913.28	54.52
15	10000	25	60.12	1.32	14.92	1834.28	95.48
20	100	13	40.54	1	10.15	1115	46.62
20	500	5	40.4	0.8	8	74.6	3.8
20	1000	5	66.4	1	12.2	133.2	9.8
20	5000	5	94.8	1.4	17.2	709	43.6
20	10000	5	73.4	1	18	1481	74.8
25	100	5	5.4	0	1	8.8	1
25	500	5	31.4	1	7	51.6	3.2
25	1000	5	40	1.4	11.8	114	7.6
25	5000	5	66	1	15.6	540	39.2
25	10000	5	112.4	1	17.8	1338.4	91.8

Table 2: Cluster sizes

We can see from this table that this K-medoids approach does not produce balanced clusters. There tend to be a few clusters that have many nodes in them, that take up a large portion of the tree and a few clusters that are very small. Our goal is to have clusters that are balanced, each having about the same number of nodes, and relatively compact with low intracluster distance. Some work will need to be done in order to achieve this goal.

6 Conclusions

From our results we can see that, as is, Method 2 is outperforming Method 1. Since the average run times for Method 2 are about the same regardless of the number of nodes and medoids we believe that this should be able to run quickly on any real districting plan data. However, we still need to ensure that the resulting clusters are relatively the same size, something that has not been achieved yet. A potential way to do this is to bring the problem closer to the real life scenario. When making a districting plan the number of voters in each district needs to be roughly equal. Thus we can assign each node a population number and take the population of each district into account when we are assigning nodes to medoids in the algorithm.