Dan Nguyen
Assignment 5
CSCI 3202
10/16/16

A Simulated Annealing Solution to Gerrymandering

Purpose

Gerrymandering is the process of drawing political boundaries in a region a certain way to give a certain political party an unfair advantage in an election. It is proven that the result of an election between two parties can be fixed based on how the political divides of the region are determined. Unfortunately, solving for all combinations of political district configurations is *computationally hard,* meaning that no algorithm has been proven to give a perfect optimal solution.

The approach to creating a solution to this problem in this assignment will be using a simulated annealing algorithm. The algorithm reflects the process of cooling iron as a metaphor because for iron to cool, it must be done slowly or else the iron will deform. In comparison to the problem, if our algorithm approaches a solution too quickly, the solution is likely not going to be as optimal as a solution that we would get if the algorithm had kept running for longer.

Our algorithm is going to find a configuration of a city, containing an n x n grid of neighborhoods. Each neighborhood is either going to be with the “Rat’s” or the “Dragon’s” political party. Our job is to create a configuration of political districts and boundaries of neighborhoods so that the election is as fair as possible and to minimize gerrymandering.

Procedure

Fitness Function

Our fitness function is a function that we have that evaluates how “good” a solution is. For us we want our function to rate how “fair” a city’s district boundary configuration is for an election. Our fitness function will represent a configuration’s level of optimality with a single number or score. It takes input parameters of type city, which has n districts. These districts have n votes in an array, indices represent a neighborhood and its political view. A district has highest fitness when #R = #D. The city’s fitness is the sum of the fitness of each district. The fitness function returns an int |#R - #D|; 0 is optimal.

Neighbor Detection

Our algorithm is going to have to come up with a bunch of random configurations of the borders, however based on the constraints of the assignment, we are going to have to discard a lot of those that don’t meet the criteria, and select from the ones that pass our validity test. Districts neighborhoods must all connect by borders left, right, up, down, and all diagonals. We had a set object for our district object that contained a map for each neighborhood. The key was a tuple that stored a value of the neighborhoods voting affiliation, i.e. {(0,1):R} would represent the neighborhood at 0, 1, and that neighborhood specifically voted for the Rats.

To detect neighboring cells of districts, we created an adjacencyMatrix in our isNeighbor() function for each neighborhood. isNeighbor() returned true or false when

comparing two neighbors. It would be true if the two nodes were in each other's adjacency matrix, which is a matrix that contains the neighbors of each neighborhood in all 8 directions.

Generating New Candidate Solutions

Making new random candidate solutions is the basis of simulated annealing. However, with the limited resource of time and planning, I did not fully implement the idea, however the theory is quite simple actually, which we will discuss. We take our initial solution. We then pick a random neighborhood, and a neighboring district's neighborhood, and if they are neighbors (from the previously mentioned function) we swap the two neighborhoods and then belong on the other's original district. We then evaluate the fitness of this new configuration. If the fitness of the new solution is more fit than the previous, then set it as the most fit. if fitness(new) > fitness(old), old = new, re loop.

Data

The program reads in .txt files that are n lines long. Each line is filled with n characters representing either "R" or "D", separated by blank spaces; " ".

largeState.txt
Party division in population:
****
R:< 52% >
D:< 48% >
****

```
D R D R D R R R D R
D D R D R R R R R R
D D D R R R R R D D
D D R R R R D R R D
R R D D D R R R D R
R D D D D D R R R R
R R R D D D D D D D
D D D D D D R D R D
R R D D R R D D R R
R R D D R R D D R R
```

smallState.txt
Party division in population:
****
R:< 50% >
D:< 50% >
****

```
D R D R D R R R
D D R D R R R R
D D D R R R R R
D D R R R R D R
R R D D D R R R
R D D D D D R R
R R R D D D D D
D D D D D D R D
```

Results

In simulated annealing, different values for temperatures will provide different results and optimal solutions. The temperature adjustment determines how cold the iron gets in the analogy of melted iron cooling. If the temperature adjustment goes up, that means that the algorithm will find a solution faster, and there is a higher chance it won't be as optimal as if it cooled slower.

The solutions "good"ness was determined on each districts fitness function, and then the city was summed up in total as well. This way we will deem a solution optimal if it is as close as possible to an equilibrium R/D split. In this manner, if we kept running the algorithm over repeatedly, we would NOT reach the same solution because our randomly generating neighbor configurations are chosen by random. Over time similar optimal solutions will repeat on the

contrary though. The amount of unique search states explored are defined by how soon the equation $e^{deltaFitness/k}$ is met and accepted.