Jake Traut

10/13/16

CSCI 3202 Assignment 5 Report

Purpose:

In this assignment we examined different methods of divvying up a population, or collection of data, into subgroups of equal size with resulting costs. In our case, the program is intended for generating political district boundaries in a given voter population with two parties, and it searches for the most fair distribution. I used the simulated annealing algorithm to accomplish this task, generating thousands of solutions (equivalently covering a large search space) and as the algorithm converged, or toned in, on the best of the solutions it finds this "fair" political district.

Procedure:

Making this algorithm work involved writing many separate functions that the algorithm calls to check all sorts of cases that we care about. For storing the population data I used a graph implementation that allowed me to add all nodes (0-63 for small state and 0-99 for large state) and keep track of adjacent or neighboring nodes (as all districts have to be contiguous, voters in a district must be "neighbors"). To check for valid districts I ran a depth first search on each district of a particular solution with an extra case of requiring an unvisited adjacent node to also be in the same district before expanding on it. This allowed me to tell if my districts were contiguous or not. I broke this sub-task into two functions being "DFS" and "isValid" were isValid calls DFS and examines the output from DFS. The other functions used include "fitness" for defining what a good solution is, "isNew" for making sure I'm not repeating the same solutions and/or counting them more than once, and "generateNeighbor" for producing a new solution from the current solution.

- **Fitness Function:**

  My fitness function evaluates how "fit" a particular solution is by tallying up the total number of 'D' voters and 'R' voters per district, and with that information adds one to the total number of 'D' districts or 'R' districts depending on the majority of voters. I then took the absolute value of the difference between these two values to get the difference in majority districts, and this is the value that I want to minimize (0 being optimal). I also kept track of "TIED" districts, when there are equal R and D voters in a district, mainly to make sure everything's being accounted for correctly when say the outcome of majority districts is 3-3 and I want to know what happened to the other 2 districts (in the case of small state). The way I kept track of what party each voter belonged to was by reading in the text from the file and storing the affiliation in a global "VOTERS" array where the index is the specific voter (0-63 or 0-99) and the value being the value read (either 'D' or 'R').

  As an example I'll use my initial solution and final solution for small state, the initial being each district is a row from the text and the final being a well-fit solution that my simulated annealing algorithm returns. I'll go into more detail with the initial as there's more simplicity in manually deciphering this one.

INITIAL SOLUTION
District 1: [0 1 2 3 4 5 6 7] -> D R D R D R R R -> 5R – 3D -> R districts + 1 = 1
District 2: [8 9 10 11 12 13 14 15] -> D D R D R R R R -> 5R – 3D -> R districts + 1 = 2
District 3: [16 17 18 19 20 21 22 23] -> D D D R R R R R -> 5R – 3D -> R districts + 1 = 3
District 4: [24 25 26 27 28 29 30 31] -> D D R R R R D R -> 5R – 3D -> R districts + 1 = 4
District 5: [32 33 34 35 36 37 38 39] -> R R D D D R R R -> 5R – 3D -> R districts + 1 = 5
District 6: [40 41 42 43 44 45 46 47] -> R D D D D D R R -> 3R – 5D -> D districts + 1 = 1
District 7: [48 49 50 51 52 53 54 55] -> R R R D D D D D -> 3R – 5D -> D districts + 1 = 2
District 8: [56 57 58 59 60 61 62 63] -> D D D D D D R D -> 1R – 7D -> D districts + 1 = 3
This produces a total of 5 R districts and 3 D districts, thus a final fitness of 2 which is not very good.

FINAL SOLUTION (random, will use the most recent run of my program)
District 1: [0 1 8 9 16 17 24 25]
District 2: [2 3 4 5 6 10 11 12]
District 3: [31 39 47 53 54 55 62 63]
District 4: [7 13 14 15 18 19 20 21]
District 5: [22 23 26 27 28 29 30 34]
District 6: [32 33 40 41 42 43 48 56]
District 7: [35 36 37 38 44 45 46 52]
District 8: [49 50 51 57 58 59 60 61]
This solution produces 3 R districts and 3 D districts (2 left as TIED) which makes for a fitness of 0 (being optimal) as no single party has domination in the elections.

- **Neighbor detection:**

  I actually didn't make an individual function dedicated to this task, but it's certainly required for producing good solutions and I included a check for neighboring districts in my new solution generator. The way I went about this was using a while loop that continues to iterate until a bordering district is found (bordering the first random district I chose). I used borders as a Boolean value and while not borders it continues to loop, and I update borders to true once I have found a second random district where a random voter in that district has an adjacent node that exists in the first district. I check for adjacency by using the Graph class and the "findVertex" function.

- **Generating new candidate solutions:**

  As I previously explained, my neighbor detection implementation falls into this function as well. In more detail, before entering my while loop that ensures finding a bordering district that boosts probability of generating a valid solution, I pick a random district from the solution passed into "generateNeighbor" and then a second random district that includes a check to make sure these aren't the same districts. I then go into the while loop, where once a neighboring district is found (checking the second against the first, so the second district may change) I continue to do more work to get an even higher chance of producing a valid solution. The preferred option at this point is finding a good node to swap from the first district to the second, so I set another Boolean "found_swap" to false and loop through all the nodes in the first district hoping to find a second node that also borders the second district (and is also not the first adjacent found). This way I have two separate bordering nodes, one from each district that I can dedicate to swapping, and at this point can set found_swap to true and generate a new solution.

If I make it out of this for loop without results, i.e. found_swap is still false; I generate a more random, less likely valid, solution. This is done by simply choosing a random node in the first district (that is not the adjacent found from district 2) and dedicating that node to the second district while accepting the definitely bordering node of the second district into the first.

To determine if the solutions being generated by my "generateNeighbor" function are indeed valid, I call two more functions in my "simAnnealing" algorithm directly after getting a possible new solution back. These functions include "isValid" and "isNew" which should be fairly self-explanatory in what they are checking, but I'll go into more depth for each.

Inside isValid I loop through each district of a solution, resetting global variables "VISISTED" and "PARENTS" to their respective default values per iteration before calling DFS with the root (min) node. DFS then populates these variables for isValid to determine validity for the specific district. In this case, validity is contiguousness within all districts, so if any of these iterations come false, then the entire solution is invalid. I check for validity by using the PARENTS dictionary, and if any of the nodes' (besides the root) parent doesn't exist in the current district, I return false.

As for isNew, I am simply checking if the recently generated solution has already occurred and been considered, such that I don't over count my search space and continue to reevaluate the same solutions over and over. This is done by using a global variable I named "OLD_SOLUTIONS" that stores every previous **valid** solution, and whatever solution I pass into isNew is then compared to every other solution in OLD_SOLUTIONS and if there is a match I return false, and if not I set OLD_SOLUTIONS[number of solutions] equal to a copy of the current solution.

If both of these tests are passed, the new solution can be further considered in the SA function before being accepted or omitted.

Data:

The provided data for this assignment included two text files, "smallState.txt" and "largeState.txt" that each contains an n x m matrix (actually n x n) of either 8 x 8 or 10 x 10, where each entry in the matrix is either a 'D' or 'R' representing the party. I converted this data into a list of node keys that have their party affiliation stored elsewhere (in an array corresponding to key values). With these values, I produced a graph for all the nodes where I stored all edges or which nodes are adjacent. Processing this data gave me the results that the small state has a 50-50 split between voters while the large state has a 52-48 majority going to the R party. I got these numbers by keeping track of the total number of D's and R's read in from the file and dividing them by the number of total voters, then multiplying by 100 to get a percent.

Results:

My simulated annealing approach produces what I deemed to be the most-fit solutions on every run. In other words, I always get a result where there exists an equal amount of D and R majority districts, whether this majority is 2-2, 3-3, or 4-4 (in terms of the small state, but also for large). This perfectly follows my definition of fitness as no party ever has the overall majority vote going into the elections. There are many different configurations of boundaries to reach this result, as I don't believe I've ever seen the exact same solution come out which isn't much of a surprise as we're working with a fairly large search space. When I run my program with initial values of T = 1.0, k = 85, alpha = .95 and Tmin = .0001, if I run while x < 100 per T iteration, I get somewhere around 2200 unique solutions in under 10 seconds, whereas if I run it with x < 1000, I get a search space of approximately 23000 in just about 6 minutes. Note these search space values vary by about a thousand in either direction for x < 100 runs and 1-4 thousand for x < 1000. I don't notice much of a change when tuning the other parameters by reasonable amounts.