Jeremy Udis
October 14th, 2016

Assignment 5 Report

Purpose:

The purpose of the program is to exhaustively search for a solution where the ratio of majority districts in the search space is as close to the ratio of the political representation of the entire population. For example, among the population, if half are apart of the dragon party and half rabbit party, then the goal is to have the ratio of the two districts be split on, or as close to that ratio.

Fitness Function

My fitness function works as follows: there is some initial start state, s0. This start state is a decimal value between 0 and 1, representing the number of rabbit districts divided by the number of districts. This works by counting the number of rabbit, dragon, and neutral states. If there are neutral districts, omit those from the calculation. For example, if there are 3 rabbit districts, 3 dragon districts, and 2 neutral districts, the percentage of rabbit districts would be ½. Then, generate neighboring solution, called sPrime. This value is calculated the same way as s0. Set s = s0. The fitness function checks which value,s(status quo) or sPrime is closer to the rabbitRatio = #ofRabbits/Population. If abs(1-sPrime/rabbitRatio) < abs(1-s0/rabbitRatio), then accept sPrime. Here is a worked example with sPrime = 0.5 and s = 0.75 rabbit ratio = 0.50:

abs(1-0.5/0.5) < (1 - 0.75/0.5) = abs(0) < abs(0.5) = 0 < 0.5. This is true, set s = sPrime

Neighbor detection

For checking neighbors, I added a property to my Simulated Annealing class called district number, which gives me the value of which district the node belongs to. For Data Structures, I used a matrix of nodes that have properties such as political affiliation and district number.

Generating new Candidate Solutions

This is where my program is most likely different from others in the class. Instead of generating random solutions and then check to see if each district is contiguous, my algorithm is what I would call smart random. The algorithm has a series of edge cases based on the configuration of the matrix. In particular, I generate random x,y coordinates where 0 <= x,y <= Width or Length of the matrix. Then, based on the coordinates, there are if statements to handle the edge and corner cases. There are four corners and four edges, so I account for all of them. For example, if x = 0 and y = 4, that would hit the top edge of the matrix. Once in the if statement, I check adjacent nodes to see if they are in different districts and their political affiliation are different. From there, There are a finite number of swaps possible if the districts must be contiguous. While diagonal nodes are considered contiguous, diagonal swaps of two nodes satisfying the criteria above of any kind are not allowed, as they leave nodes behind. Take x = 5, y = 5 on a 10x10 matrix. This is the general case, as there are possible coordinates that surround the location. From this point, there are a maximum of 4 possible swaps that satisfy the conditions above(x:5 y:6, x:6 y:5, x:4 y:5, x:5 y:4). If all conditions are satisfied, I swap the corresponding values, which are stored in a dictionary.

Data

The data used in the program was derived from a .txt file. The file contained a matrix of D’s and R’s, each representing a different political affiliation. For the smallState.txt, the percentage from each party was split 50 50, while the largeState.txt was %52 rabbits | %48 dragons.

Results

The results of the program was interesting considering that the result is not always the same. Since the program is somewhat random, along with the fact that the program may not always find the optimal solution based on the input; it’s easy to imagine there is room for interpretation. As far as parameters, as T*k would decrease, the possibility of accepting the local non-optimum would decrease, and the bigger the T, the more iterations run. The solutions were optimal based on the input, meaning that they were not always optimal. However, the algorithm result was always an improvement over the start state, and in many cases, was optimal or very close to optimal. Because of my method for generating smart search states, my algorithm only needed explore ~1000 unique/valid solutions to reach an optimal or near optimal solution.