Jeffrey Lipnick

CSCI 3202 – Introduction to Artificial Intelligence

Instructor: Hoenigman

Assignment 5

**Report**

**Purpose**

The purpose of this program is to find a "good" solution for a political redistricting problem. A solution is defined as "good" based on a fitness function within the simulated annealing algorithm. Simulated annealing is used due to the large search space of the problem, making it impossible to find the optimal solution.

For this specific problem a good solution would provide an even number of districts with both a Rabbit and Dragon majority. Alternatively, a good solution would have many districts where the number of Rabbit and Dragon voters are tied. The idea is to create a districting scheme that accurately represents the voting population. For example, smallState.txt has an even number of Rabbit and Dragon voters. A good solution would have the district split be as even as possible. When considering largeState.txt there are slightly more Rabbit voters, so a good solution would slightly favor the Rabbit voters in terms of district majorities.

**Procedure**

**Fitness Function**

For every district in the solution, the fitness function calculated the number of Rabbit voters and Dragon voters. These numbers were subtracted, the absolute value taken, and then the absolute value was added to a total. The total was then divided by the number of districts to provide an average. The final number is the average of the difference between Rabbit voters and Dragon voters in a district.

Some example inputs (using only smallState.txt for the sake of simplicity):

-If every district had an even number of Rabbit and Dragon voters (4 each), then the fitness function would return a 0. This would be an optimal solution.

-Due to the constraints of the problem, the next most optimal situation is where there are 5 voters of one party and 3 of another party. This is because each district needs to have 8 voters. This would return a value of 2 for that district. Let’s say that every district had this sort of distribution, then the value would be (2 \* 8) / 8. The fitness function would return a value of 2 for this solution.

**Generating new Candidate Solutions**

To generate a new candidate solution, I used a while loop since it is possible that invalid solutions could be generated. Here are steps through the algorithm:

1. A random district is selected. We’ll call this district1
2. A random node is selected from that district. We’ll call this node.
3. Node’s adjacent nodes are looped through until one is found from a different district. If there are no adjacent nodes from a different district, then no more code is executed and the while loop restarts.
4. The adjacent node is added to district1 and removed from it’s current district (we’ll call this district2).
5. A node is randomly selected from district2 (as long as it isn’t the original adjacent node).
6. This node’s adjacent nodes are looped though in search of a node in district1. If no viable nodes are found, a new node in district2 is selected instead. Once a viable node in district1 is selected, the algorithm continues. We’ll call this selected node adj.
7. Adj is added to district2 and removed from district1.
8. A depth-first search is performed on district1 and district2 to make sure the solution is contiguous. If it is not, then the main while loop restarts, otherwise the solution is returned in the simulated annealing algorithm for fitness evaluation.

**Neighbor Detection**

After smallState.txt and largeState.txt are loaded into the program as a matrix, that matrix is looped through to set up the adjacent nodes for every node. Every node in the matrix has a dictionary of adjacent nodes. These values are fixed (since the voters aren’t moving), and are populated as follows:

Python allows for negative indexing of lists, so great care needed to be taken so that all the positioning cases were appropriately handled. For example, the node in position (0,0) in the matrix couldn’t have neighbors above or to the left. Nested loops were used to go through every node in the matrix and populate their adjacent nodes. The top/bottom rows, left/right columns, and 4 corners were special cases that needed to be handled with appropriate checks.

The solution generation and checking for contiguity were made much simpler by having the adjacent nodes already populated for every node in the matrix.

**Data**

The data used in this program were 2 text files, "smallState.txt" and "largeState.txt". Each text file contained a matrix of the letters D and R to represent voters for the "Dragon" and "Rabbit" political parties. smallState.txt is an 8 x 8 matrix, and largeState.txt is a 10 x 10 matrix. The program could be passed 1 file at a time as a command line argument to run the simulated annealing algorithm on.

smallState.txt had 32 Dragon voters and 32 Rabbit voters. 50% were Dragon voters and 50% were Rabbit voters.

largeState.txt had 48 Dragon voters and 52 Rabbit voters. 48% were Dragon voters and 52% were Rabbit voters.

**Results**

In general, the simulated annealing algorithm produced roughly an even number of majority districts, with some districts being tied. The main parameter I tuned in the probability acceptance portion of the algorithm was the value for k. I kept T, Tmin, and alpha the same, and used k to change how quickly the probability of acceptance for a worse solution decreased. In the probability acceptance calculation I used, the smaller the value of k, the more quickly worse solutions would not be accepted. If I kept k too large, then it would accept poor solutions all the way through the end of the simulated annealing run. I found that a value of 50 for k produced the most evenly distributed districts of the values I tested.

The results could be improved with a more sophisticated fitness function. The fitness function I used could have been improved to provide more granular feedback about the quality of a solution. Perhaps this could have been accomplished by comparing the ratio of voters in a district to the overall ratio of voters in the total population.

Each run of the algorithm produced different results. Although, the results tended to have an equal or closely equal number of majority districts for Rabbits and Dragons. If the fitness function were improved, the diversity of solutions would likely decrease as the algorithm would be more discerning regarding what constituted an improved solution.

660 unique search states were explored. This can easily be increased or decreased by adjusting Tmin.