Redistricting Recursion Project

Sergio Cardenas

December 5, 2019

1. Background

The way the people of the United States are represented in the Legislative branch of the Government is by voting for a representative who will vote in our interests on Congress. A citizen can only vote for a candidate who is running on his/her district. Said district's boundaries are determined by the state's legislators using data from the Census. There is a method known as redistricting, or gerrymandering, which can distribute voters between districts in a way that will dilute the vote of a party. This can be done to the extent and with the intention of having the majority of a population have the minority representation in Congress.

Redistricting is done through two methods, packing and cracking. Packing is when you put as many voters of one party as you can into a single district. This dilutes their vote because to win a district race the representative needs 51% of the vote, so when you pack a district so that the population of that districts votes, as an example, 95% for that candidate, then 44% of that district's votes are wasted since they were used on a representative that didn't need them. The other method, known as cracking, is when you spread voters of one party thinly into multiple districts so that they can't be the majority in any of their districts. Performing these two methods strategically is what makes gerrymandering possible.

2. Methodology

The purpose of my project is to develop a code that will take as parameters a population and the way their votes are split to distribute the voters between a number of districts in a way that the party that had the least number of votes will have the majority in more districts. We need to state an assumption and a simplification for this process. The assumption is that we know how each person votes, and the simplification is that we are not consider geographical boundaries.

The way we approach this is first by determining that the party that has the minority has "-1" voters, and that the party that has the majority has "+1" voters. To explain the methodology, we take into consideration this example: We want to split 15 voters, 9 of whom vote "+1" and 6 of whom vote "-1" into 5 districts. Each vote is visualized as one box, and each district is visualized into a row of boxes. The fair distribution of the 15 voters into 5 districts is shown in Figure 1. This distribution gives 3 out of the 5 districts a majority of "+1" voters and therefore 3 out of the 5 representatives sent to Congress from this population will be voting "1".

District 1	-1	-1	-1
District 2	-1	-1	-1
District 3	+1	+1	+1
District 4	+1	+1	+1
District 5	+1	+1	+1

Figure 1: 15 voters split into 5 rows, or 3 districts.

We define the variable “n” and the variable “max”, as the total number of districts we are working with and the maximum number of voters per district, which is computed as the total population divided by the total number of districts and rounded down. We start distributing the “-1” voters from the fist district until the district (n-1), or in this case district 4, until all “-1” are used as shown in Figure 2(a). Then, if the districts haven’t reached their max value, 3, then we will those districts with “+1” voters until each district reached in size its max value, as shown in Figure 2(b). What we just did here is cracking the “+1” votes. Finally, all the leftover “+1” voters are added to the n district, or in this case district 5, as shown in Figure 2(c). What we just did here is packing the “+1” votes.

a)

District 1	-1	-1	
District 2	-1	-1	
District 3	-1		
District 4	-1		
District 5			

b)

District 1	-1	-1	+1
District 2	-1	-1	+1
District 3	-1	+1	+1
District 4	-1	+1	+1
District 5			

c)

District 1	-1	-1	+1
District 2	-1	-1	+1
District 3	-1	+1	+1
District 4	-1	+1	+1
District 5	+1	+1	+1

Figure 2: Redistricting Through Three Steps.

Our final redistricted population, as shown in Figure 2(c), has 3 districts, districts 3, 4, and 5 with a majority of “+1” out of the 5 districts. In other words, our redistricting didn’t give the minority vote “-1” the majority in the most districts. When this happens, we must do the whole process again but now for districts 1 to 4 while leaving district five intact. This is our recursion progress. Now the variable n is 4 instead of 5, we again distribute our “-1” votes through districts 1 to district (n-1), or in this case 3, as shown in Figure 3(a). Then we fill up districts 1 through 3 with “+1” votes until they reach their max value of 3 which doesn’t change

each recursion, as shown in Figure 3(b). Finally, we fill up our nth district, district 4, with the remainder "+1" votes, as shown in Figure 3(c).

a)

District 1	-1	-1	
District 2	-1	-1	
District 3	-1	-1	
District 4			
District 5	+1	+1	+1

b)

District 1	-1	-1	+1
+District 2	-1	-1	+1
District 3	-1	-1	+1
District 4			
District 5	+1	+1	+1

c)

District 1	-1	-1	+1
District 2	-1	-1	+1
District 3	-1	-1	+1
District 4	+1	+1	+1
District 5	+1	+1	+1

Figure 3: Recursive Redistricting Through Three Steps

Our final redistricted population, as shown in Figure 3(c), gives three out of the five districts a majority vote of "-1", and therefore gives the minority vote a higher representation in Congress. With this population, we were able to gerrymander the population with two recursions. If the districts in Figure 3(c) wouldn't have given a majority vote of "-1" in the majority of districts, then we would have done the process again this time with for districts 1 through 3 with districts 4 and 5 kept intact, and so on and so forth until we developed a gerrymandered state or determined that this specific set of votes and districts can't be gerrymandered.

3. Code Explanation

Before starting to explain my code, I need to address a couple of comments. My code will always assume that the "+1" vote population is higher than the "-1" vote. My code works best when the population is split evenly between districts. Otherwise my code will occasionally stack aggressively one district disproportionally, depending on the numbers. Finally, working with classes and methods was rather difficult since methods can't output in vectors and all my data inputs and needed outputs are in vectors. Therefore, I worked with a function that was split with

comments that helped my guide the extensive code. This was the way I approached my code and it was the best way I could visualize and guide myself.

The main block of code calls a function that will take as parameters the population vector, a vector of how many voters voted "+1", a vector of how many voters voter "-1", the number of districts, a number of districts that won't change with each recursion, and a counter that will determine how many times the recursive function keeps calling itself:

```
int main(){
        int pop, avot, bvot, districts;
        cout << "Enter population size: " ; cin>>pop; cout<<endl;
        cout<< "Enter number of +1 voters: " ; cin>> avot; cout<<endl;
        cout<< "Enter number of districts: " ; cin>> districts; cout << endl;
        vector<int> population(pop);
        vector<int> avoters(avot,1);
        vector<int> bvoters(bvot,-1);
        int total = districts;
        int count = 0;
        minority_rules(population,avoters,bvoters,districts,total,count);
        return 0;
};
```

The 1st block of code of the function's purpose is to distribute "-1" votes for districts 1 through (n-1) until we run out of said votes:

```
        for (i=0;i<100;i++){

                if (i==val){i=0;};
                matrix.at(i).push_back(bpop.at(j));
                j=j+1;
                if(j==bpop.size()){break;};
        };
```

The 2nd block of the code distributes "+1" votes on the condition that districts 1 through (n-1) aren't the same size:

```
int count =0; int l = 0 ;
if (i!=(val-1)){
        for (k=i;k<val;k++){
                matrix.at(k).push_back(apop.at(l));
                l=l+1;
                count = count + 1;
        };
};
```

The 3rd block of the code distributes "+1" votes for districts 1 though (n-1) until the districts reach their maximum capacity:

```
int account=0;
for (int i=0;i<100;i++){
        if (i==val){i=0;};
        matrix.at(i).push_back(apop.at(l));
        l=l+1;
        acount = acount +1;
        if(matrix.at(val-1).size()==max){break;};

};
```

The 4th block distributes the remaining "+1" votes that haven't been used on districts 1 to (n-1) to district n:

```
for (int k=0; k<100 && k<(apop.size() - count - acount);k++){
        matrix.at(val).push_back(apop.at(k));
```

The 5th block for each district determines the majority of votes and then make a vector that will have the majority of each district as a member:

```
for (int j=0; j<n;j++){
        vector<int> avote, bvote;
        for (int i=0; i<matrix.at(j).size(); i++){
                if (matrix.at(j).at(i)==1){avote.pushback(1);
                if (matrix.at(j).at(i)==-1){bvote.pushback(-1);
        };
if(avote.size() > bvote.size()) {lean.at(j)==1;}
else {lean.at(j)==-1;};
```

```
};
```

The 6th block determines whether more districts voted for "+1" party or for the "-1" party from the vector created in the previous block. This block also needs to consider what vectors where put aside in previous iterations of the function, and that is why we have the counter parameter:

```
vector<int> alean,blean; int major;
for (int j=0; j<n; j++){
        if(leant.at(j)==1){alean.push_back(1);};
        if(leant.at(j)==-1){blean.push_back(-1);};
};
if(blean.size() > (alean.size()+Counter)) {major=-1;}
else {major=1;};
```

The 7th and final block is our recursive portion. If the majority of districts vote "-1", then the vector of redistricted voters is printed and the function ends. If the majority of districts vote "+1", then the code will develop a new vector of the population, a new vector of the population that votes "+1", and a new vector of the population that votes "-1" without taking into account the "+1" votes that were distributed to the n district. It will add one to the counter as a parameter, and then it will call the recursive function again with the new vectors, (n-1), the total number of districts that doesn't change with the recursions, and the counter that determines how many times the function has been called back :

```
if (major==-1) {
        for (int i=0; i<n; i++){
                for (int j=0; j<matrix.at(i).size();j++){
                        cout<< matrix.at(i).at(j)<< " ";
                };
                cout << endl;
        };
        cout << "Out of "<<all<<" districts, "<< bean.size<<" now leans -1"<<endl;
        return;
};
if(major==1){
```

```
        for (int j=0; j<matrix.at(val).size();j++){
                cout<< matrix.at(val).at(j)<< " ";
        };
        cout<<endl;
        vector<int> ARRAY, APOP, BPOP;
        for (int i=0; i<val; i++){
                for (int j=0; j<matrix.at(i).size();j++){
                        if (matrix.at(i).at(j)==1){APOP.pushback(1);};
                        if (matrix.at(i).at(j)==-1){BPOP.pushback(-1);};
                };
        };
        Counter=Counter+1;
        Return minority_rules(ARRAY,APOP,BPOP,n-1,all,Counter);
};
```

4. Code Results:

- Population: 15; +1 Voters: 9; -1 Voters: 6; Districts: 5

1	1	1
-1	-1	1
-1	-1	1
-1	-1	1
1	1	1

Out of 5 districts, 3 now leans -1

- Population: 24; +1 Voters: 14; -1 Voters: 10; Districts: 3

-1	-1	-1	-1	-1	1	1	1
-1	-1	-1	-1	-1	1	1	1
1	1	1	1	1	1	1	1

Out of 3 districts, 2 now leans -1

- Population: 40; +1 Voters: 25; -1 Voters: 15; Districts: 8

1	1	1	1	1

1	1	1	1	1
-1	-1	-1	1	1
-1	-1	-1	1	1
-1	-1	-1	1	1
-1	-1	-1	1	1
-1	-1	-1	1	1
1	1	1	1	1

Out of 8 districts, 5 now leans -1

Note: The way Texas is gerrymandered isn't to give the Republican party the majority in the House of Representatives since they already have it. Texas was gerrymandered so that the Democratic party instead of representing 47.22% of the seats in the 2018 election, they only represented 36.22%. My code only works to give majority representation to the minority of the party. Texas was gerrymandered so that the majority could represent a super majority in Congress.

5. Overall Code

```
#include <iostream>
#include <vector>
using std::cout; using std::endl; using std:: using std::vector; using std::cin;
void minority_rules(vector<int> array, vector<int> apop, vector<int> bpop, int n, int all, int &
int Counter){
        vector<vector<int>> matrix(n);vector<int> lean(n);
        int max= array.size()/n; int i = 0; int j = 0; int val = (n-1);
        //separate the population into given districs
        //1- spreads B votes through n-1 districts
        for (i=0;i<100;i++){
                if (i==val){i=0;};
                matrix.at(i).push_back(bpop.at(j));
                j=j+1;
                if(j==bpop.size()){break;};
```

```
};
//2- if the districts aren't the same size, fill them up with a votes
int count =0; int l = 0 ;
if (i!=(val-1)){
        for (k=i;k<val;k++){
                matrix.at(k).push_back(apop.at(l));
                l=l+1;
                count = count + 1;
        };
};
//3- fill up all n-1 districts until they reach maximum capacity
int account=0;
for (int i=0;i<100;i++){
        if (i==val){i=0;};
        matrix.at(i).push_back(apop.at(l));
        l=l+1;
        acount = acount +1;
        if(matrix.at(val-1).size()==max){break;};
};
//4- fill up the final n district with the leftover 1 votes
for (int k=0; k<100 && k<(apop.size() - count - acount);k++){
        matrix.at(val).push_back(apop.at(k));
};
// lean of each district
for (int j=0; j<n;j++){
        vector<int> avote, bvote;
        for (int i=0; i<matrix.at(j).size(); i++){
                if (matrix.at(j).at(i)==1){avote.pushback(1);
                if (matrix.at(j).at(i)==-1){bvote.pushback(-1);
        };
if(avote.size() > bvote.size()) {lean.at(j)==1;}
else {lean.at(j)==-1;};
};
 // overall lean of population
vector<int> alean,blean; int major;
for (int j=0; j<n; j++){
        if(leant.at(j)==1){alean.push_back(1);};
        if(leant.at(j)==-1){blean.push_back(-1);};
};
if(blean.size() > (alean.size()+Counter)) {major=-1;}
else {major=1;};
// recursion
if (major==-1) {
        for (int i=0; i<n; i++){
```

```
                        for (int j=0; j<matrix.at(i).size();j++){
                                cout<< matrix.at(i).at(j)<< " ";
                        };
                        cout << endl;
                };
                cout << "Out of "<<all<<" districts, "<< bean.size<<" now leans -1"<<endl;
                return;
        };
        if(major==1){
                for (int j=0; j<matrix.at(val).size();j++){
                        cout<< matrix.at(val).at(j)<< " ";
                };
                cout<<endl;
                vector<int> ARRAY, APOP, BPOP;
                for (int i=0; i<val; i++){
                        for (int j=0; j<matrix.at(i).size();j++){
                                if (matrix.at(i).at(j)==1){APOP.pushback(1);};
                                if (matrix.at(i).at(j)==-1){BPOP.pushback(-1);};
                        };
                };
                Counter=Counter+1;
                Return minority_rules(ARRAY,APOP,BPOP,n-1,all,Counter);
        };
};

int main(){
        int pop, avot, bvot, districts;
        cout << "Enter population size: " ; cin>>pop; cout<<endl;
        cout<< "Enter number of +1 voters: " ; cin>> avot; cout<<endl;
        cout<< "Enter number of districts: " ; cin>> districts; cout << endl;
        vector<int> population(pop);
        vector<int> avoters(avot,1);
        vector<int> bvoters(bvot,-1);
        int total = districts;
        int count = 0;
        minority_rules(population,avoters,bvoters,districts,total,count);
        return 0;
};
```