Programming project: Redistricting

Victor Eijkhout

2020
Abstract

The US electoral system for the house of representatives groups ‘census districts’ into ‘electoral districts’ that supply a representative. Grouping the census districts differently into electoral districts can greatly affect the number of representatives the parties send to the house.

This project lets a student explore the phenomenon of redistricting, and especially ‘gerrymandering’: applying the redistricting in a way that maximally benefits one party. In particular, students will at first aim to maximize the number of representatives in a state, from a party that numerically is in the minority. Students can then explore mitigating this phenomenon.

The project description explicitly targets an object-oriented object, where creating and copying objects is simple.

This project can be done by one or two undergraduate students, or AP high schoolers at the end of a first or semester programming course.
<table>
<thead>
<tr>
<th>Summary</th>
<th>Develop a simple model or redistricting, and the potential unfairness of the process.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Topics</td>
<td>Recursion, arrays, classes&lt;br&gt;There is a possibility of exploring memoization and dynamic programming</td>
</tr>
<tr>
<td>Audience</td>
<td>Undergraduate or AP high school</td>
</tr>
<tr>
<td>Difficulty</td>
<td>High</td>
</tr>
<tr>
<td>Strengths</td>
<td>Explores a question of great societal impact.</td>
</tr>
<tr>
<td>Weaknesses</td>
<td>Requires careful analysis of the problem. No graphics output, so the interpretation of output requires some imagination</td>
</tr>
<tr>
<td>Dependencies</td>
<td>Strictly speaking no dependencies. Familiarity with the concepts of search and optimization may be helpful.</td>
</tr>
<tr>
<td>Variants</td>
<td>Expansion beyond the simple model is possible but requires considerable programming. Use of actual census data is very hard due to data formats used.</td>
</tr>
</tbody>
</table>
Chapter 1

Redistricting

In this project you can explore ‘gerrymandering’, the strategic drawing of districts to give a minority population a majority of districts\(^1\).

1.1 Basic concepts

We are dealing with the following concepts:

- A state is divided into census districts, which are given. Based on census data (income, ethnicity, median age) one can usually make a good guess as to the overall voting in such a district.
- There is a predetermined number of congressional districts, each of which consists of census districts. A congressional district is not a random collection: the census districts have to be contiguous.
- Every couple of years, to account for changing populations, the district boundaries are redrawn. This is known as redistricting.

There is considerable freedom in how redistricting is done: by shifting the boundaries of the (congressional) districts it is possible to give a population that is in the overall minority a majority of districts. This is known as gerrymandering.

To do a small-scale computer simulation of gerrymandering, we make some simplifying assumption.

- First of all, we dispense with census district: we assume that a district consists directly of voters, and that we know their affiliation. In practice one relies on proxy measures (such as income and education level) to predict affiliation.
- Next, we assume a one-dimensional state. This is enough to construct examples that bring out the essence of the problem:
  
  Consider a state of five voters, and we designate their votes as AAABB. Assigning them to three (contiguous) districts can be done as AAA | B | B, which has one ‘A’ district and two ‘B’ districts.
- We also allow districts to be any positive size, as long as the number of districts is fixed.

\(^1\) This project is obviously based on the Northern American political system. Hopefully the explanations here are clear enough. Please contact the author if you know of other countries that have a similar system.
1. Redistricting

1.2 Basic functions

1.2.1 Voters

We dispense with census districts, expressing everything in terms of voters, for which we assume a known voting behaviour. Hence, we need a Voter class, which will record the voter ID and party affiliation. We assume two parties, and leave an option for being undecided.

Exercise 1.1. Implement a Voter class. You could for instance let \( \pm1 \) stand for A/B, and 0 for undecided.

Code:

```cpp
cout << "Voter 5 is positive:" << endl;
Voter nr5(5,+1);
cout << nr5.print() << endl;
/* ... */
cout << "Voter 6 is negative:" << endl;
Voter nr6(6,-1);
cout << nr6.print() << endl;
/* ... */
cout << "Voter 7 is weird:" << endl;
Voter nr7(7,3);
cout << nr7.print() << endl;
```

Output

\[ \text{gerry} \] voters:

Voter 5 is positive:

5:+

Voter 6 is negative:

6:-

Voter 7 is weird:

Illegal affiliation value: 3

Error in creating voter 7

missing snippet voterneg  missing snippet voterwrong

1.2.2 Populations

Exercise 1.2. Implement a District class that models a group of voters.

- You probably want to create a district out of a single voter, or a vector of them. Having a constructor that accepts a string representation would be nice too.
- Write methods majority to give the exact majority or minority, and lean that evaluates whether the district overall counts as A part or B party.
- Write a sub method to creates subsets.

\[ \text{District District::sub(int first, int last);} \]

- For debugging and reporting it may be a good idea to have a method

\[ \text{string District::print();} \]
1.2. Basic functions

Code:

```cpp
cout << "Making district with one B voter" << endl;
Voter nr5(5, +1);
District nine(nr5);
cout << ". size: " << nine.size() << endl; // lean: 1
cout << ". lean: " << nine.lean() << endl;
/* ... */
cout << "Making district ABA" << endl;
District nine(vector<Voter> { {1, -1}, {2, +1}, {3, -1} });
cout << ". size: " << nine.size() << endl;
cout << ". lean: " << nine.lean() << endl;
```

Output

```
[ger] district:
Making district with one B voter
.. size: 1
.. lean: 1
Making district ABA
.. size: 3
```

Exercise 1.3. Implement a Population class that will initially model a whole state.

Code:

```cpp
string pns("-++--");
Population some(pns);
cout << "Population from string " << pns << endl; // size: 5
cout << ". size: " << some.size() << endl; // lean: -1
cout << ". lean: " << some.lean() << endl;
Population group=some.sub(1, 3);
cout << "sub population 1--3" << endl;
cout << ". size: " << group.size() << endl;
cout << ". lean: " << group.lean() << endl;
```

Output

```
[ger] population:
Population from string -++--
.. size: 5
.. lean: -1
sub population 1--3
.. size: 2
.. lean: 1
```

In addition to an explicit creation, also write a constructor that specifies how many people and what the majority is:

```cpp
Population(int population_size, int majority, bool trace=false)
```

Use a random number generator to achieve precisely the indicated majority.

1.2.3 Districting

The next level of complication is to have a set of districts. Since we will be creating this incrementally, we need some methods for extending it.

Exercise 1.4. Write a class Districting that stores a vector of District objects. Write size and lean methods:
1. Redistricting

Code:
```cpp
cout << "Making single voter population B" << endl;
Population people(vector<Voter>{Voter(0,+1)}); // size: 1
cout << ".. size: " << people.size() << endl;
cout << ".. lean: " << people.lean() << endl;

Districting gerry;
cout << "Start with empty districting:" << endl;
cout << ".. number of districts: " << gerry.size() << endl;

// Add one B voter:
gerry = gerry.extend_with_new_district(people.at(0));
cout << ".. number of districts: " << gerry.size() << endl;
cout << ".. lean: " << gerry.lean() << endl;

gerry = gerry.extend_last_district(Voter(1,-1));
gerry = gerry.extend_last_district(Voter(2,-1));

// Add two B districts:
gerry = gerry.extend_with_new_district(Voter(3,+1));
gerry = gerry.extend_with_new_district(Voter(4,+1));

// Create the population:
```

Output
```cpp
[gerry] gerryempty:
Making single voter population B
.. size: 1
.. lean: 1
Start with empty districting:
.. number of districts: 0
```

Exercise 1.5. Write methods to extend a `Districting`:
```cpp
cout << "Add one B voter:" << endl;
gerry = gerry.extend_with_new_district(people.at(0));
```

1.3 Strategy

Now we need a method for districting a population:

```cpp
Districting Population::minority_rules(int ndistricts);
```

Rather than generating all possible partitions of the population, we take an incremental approach (this is related to the solution strategy called `dynamic programming`):

- The basic question is to divide a population optimally over \(n\) districts;
- We do this recursively by first solving a division of a subpopulation over \(n-1\) districts,
- and extending that with the remaining population as one district.

This means that you need to consider all the ways of having the ‘remaining’ population into one district, and that means that you will have a loop over all ways of splitting the population, outside of your recursion; see figure 1.1.

- For all \(p = 0, \ldots, n-1\) considering splitting the state into \(0, \ldots, p - 1\) and \(p, \ldots, n-1\).
- Use the best districting of the first group, and make the last group into a single district.
- Keep the districting that gives the strongest minority rule, over all values of \(p\).

You can now realize the above simple example:
1.3. Strategy

Figure 1.1: Multiple ways of splitting a population

```
AAABB => AAA|B|B
```

Exercise 1.6. Implement the above scheme.

**Code:**

```cpp
Population five("++++-"); cout << "Redistricting population: " << endl
  << five.print() << endl; cout << ". . majority rule: "
  << five.rule() << endl; int ndistricts(3); auto gerry = five.minority_rules(ndistricts);
  cout << gerry.print() << endl; cout << ". . minority rule: "
  << gerry.rule() << endl;
```

**Output**

```
[gerry] district5:
Redistricting population: [0:+,1:+,2:+,3:-,4:-,]
.. majority rule: 1
[3[0:+,1:+,2:+,],[3:-,],[4:-,],]
.. minority rule: -1
district with one voter
```

Note: the range for $p$ given above is not quite correct: for instance, the initial part of the population needs to be big enough to accommodate $n - 1$ voters.
1. Redistricting

**Exercise 1.7.** Test multiple population sizes; how much majority can you give party B while still giving party A a majority.

1.4 Efficiency: dynamic programming

If you think about the algorithm you just implemented, you may notice that the districtings of the initial parts get recomputed quite a bit. A strategy for optimizing for this is called memoization.

**Exercise 1.8.** Improve your implementation by storing and reusing results for the initial sub-populations.

In a way, we solved the program backward: we looked at making a district out of the last so-many voters, and then recursively solving a smaller problem for the first however-many voters. But in that process, we decided what is the best way to assign districts to the first 1 voter, first 2, first 3, et cetera. Actually, for more than one voter, say five voters, we found the result on the best attainable minority rule assigning these five voters to one, two, three, four districts.

The process of computing the ‘best’ districting forward, is known as dynamic programming. The fundamental assumption here is that you can use intermediate results and extend them, without having to reconsider the earlier problems.

Consider for instance that you’ve considered districting ten voters over up to five districts. Now the majority for eleven voters and five districts is the minimum of

- ten voters and five districts, and the new voter is added to the last district; or
- ten voters and four districts, and the new voter becomes a new district.

**Exercise 1.9.** Code a dynamic programming solution to the redistricting problem.

1.5 Extensions

The project so far has several simplifying assumptions.

- Congressional districts need to be approximately the same size. Can you put a limit on the ratio between sizes? Can the minority still gain a majority?

**Exercise 1.10.** The biggest assumption is of course that we considered a one-dimensional state. With two dimensions you have more degrees of freedom of shaping the districts. Implement a two-dimensional scheme; use a completely square state, where the census districts form a regular grid. Limit the shape of the congressional districts to be convex.

The efficiency gap is a measure of how ‘fair’ a districting of a state is.
Exercise 1.11. Look up the definition of efficiency gap (and ‘wasted votes’), and implement it in your code.
1. Redistricting
Chapter 2

Style guide for project submissions

_The purpose of computing is insight, not numbers._ (Richard Hamming)

Your project writeup is at least as important as the code. Here are some common-sense guidelines for a good writeup. However, not all parts may apply to your project. Use your good judgement.

**Style**  
First of all, observe correct spelling and grammar. Use full sentences.

**Completeness**  
Your writeup needs to have the same elements as a good paper:
- Title and author, including EID.
- A one-paragraph abstract.
- A bibliography at the end.

**Introduction**  
The reader of your document need not be familiar with the project description, or even the problem it addresses. Indicate what the problem is, give theoretical background if appropriate, possibly sketch a historic background, and describe in global terms how you set out to solve the problem, as well as your findings.

**Code**  
Your report should describe in a global manner the algorithms you developed, and you should include relevant code snippets. If you want to include full listings, relegate that to an appendix: code snippets should only be used to illustrate especially salient points.

Do not use screen shots of your code: at the very least use a `verbatim` environment, but using the `listings` package (used in this book) is very much recommended.

**Results and discussion**  
Present tables and/or graphs when appropriate, but also include verbiage to explain what conclusions can be drawn from them.

You can also discuss possible extensions of your work to cases not covered.

**Summary**  
Summarize your work and findings.
2. Style guide for project submissions
Chapter 3

Bibliography