COE 322 Redistricting Project

Cameron Cummins and Conor Donihoo

December 2020

1 Introduction

Drawing districts in the United States has remained a contentious issue since its inception. Conflict arises from the act of ”redistricting” that occurs about every 10 years (in alignment with the census) when the legislative body convenes to create a map that outlines the number of districts, how many voters exist in each one, and where the borders lie. In an ideal world, the resulting map would produce a representative body that perfectly reflects the will of the people. In other words, each political party would receive a legislative representation equal to the proportion of popular votes it received in the election. It is not uncommon for a political candidate to assume office despite losing the popular vote. This same phenomenon occurs at the state level with specific political parties often receiving more representation with less popular support. Since this same legislative body dictates how districts are distributed across the state, it is common and has been openly admitted that the party in power redraws the districts in their favor. This is known as ”gerrymandering.”

In an effort to understand and eventually solve the issue of gerrymandering, we have developed a simulation and analysis of the redistricting process. We sought to not only illustrate how gerrymandering can occur, but also provide insight into ways that it could be prevented. Our project provides a robust platform for analyzing a generated or manually inputted population, drawing the districts either manually or following an algorithm, and compiling the data into a readable form.

2 Inputs and Outputs

Our project uses two dimensional arrays to perform the majority of its calculations. It takes a population in as its input and outputs a map of that population within a set of districts. It then analyzes these districts and counts the number of votes produced by each district.

The inputs for the project are as follows:

- **Population Distribution** - a two dimensional array with each cell containing a population of voters. The position of each cell is relative to the geographic position of the area of voters it represents. This array can either be manually created from a survey or generated via one of the two algorithms discussed in section 4. These cells define the resolution of the analysis, with more cells producing a higher fidelity. See section 3.3 for more detail on population cells.

- **Party Distribution** - a set of parties, with the number of occurrences of each party in the set corresponding to the desired distribution of affiliated voters in the population. This is only applicable when generating the population as opposed to using a surveyed set. See section 3.3 for more information.
- **Draw Algorithm** - the algorithm to use for drawing the districts. See more details on the algorithms available in section 5.

The outputs for the project are as follows:

- **District Distribution** - a two dimensional map of the districts relative to the population cells either inputted or generated. This output is discussed in section 3.4.
- **Districts** - vector of districts containing all voters in each district and other information relevant to that district.
- **Population Distribution** - if the distribution is generated, it is also outputted as a two dimensional array for comparison analysis.

These outputs are achieved by using the following classes and functions:

- **PopulationCell** - see section 3.3
- **Districting** - see section 3.4
- **Districting::outputDistricting()** - outputs the district ID tags for each population cell in the grid to a text file
- **genPopGridUniformRandom()** - see section 4.1
- **genPopGridUrbanCenter()** - see section 4.2
- **outputPopulationGridLean()** - outputs the first character of each population cell's party affiliation in a grid to a text file
- **outputPopulationGridSize()** - outputs the number of voters in each population cell in a grid to a text file

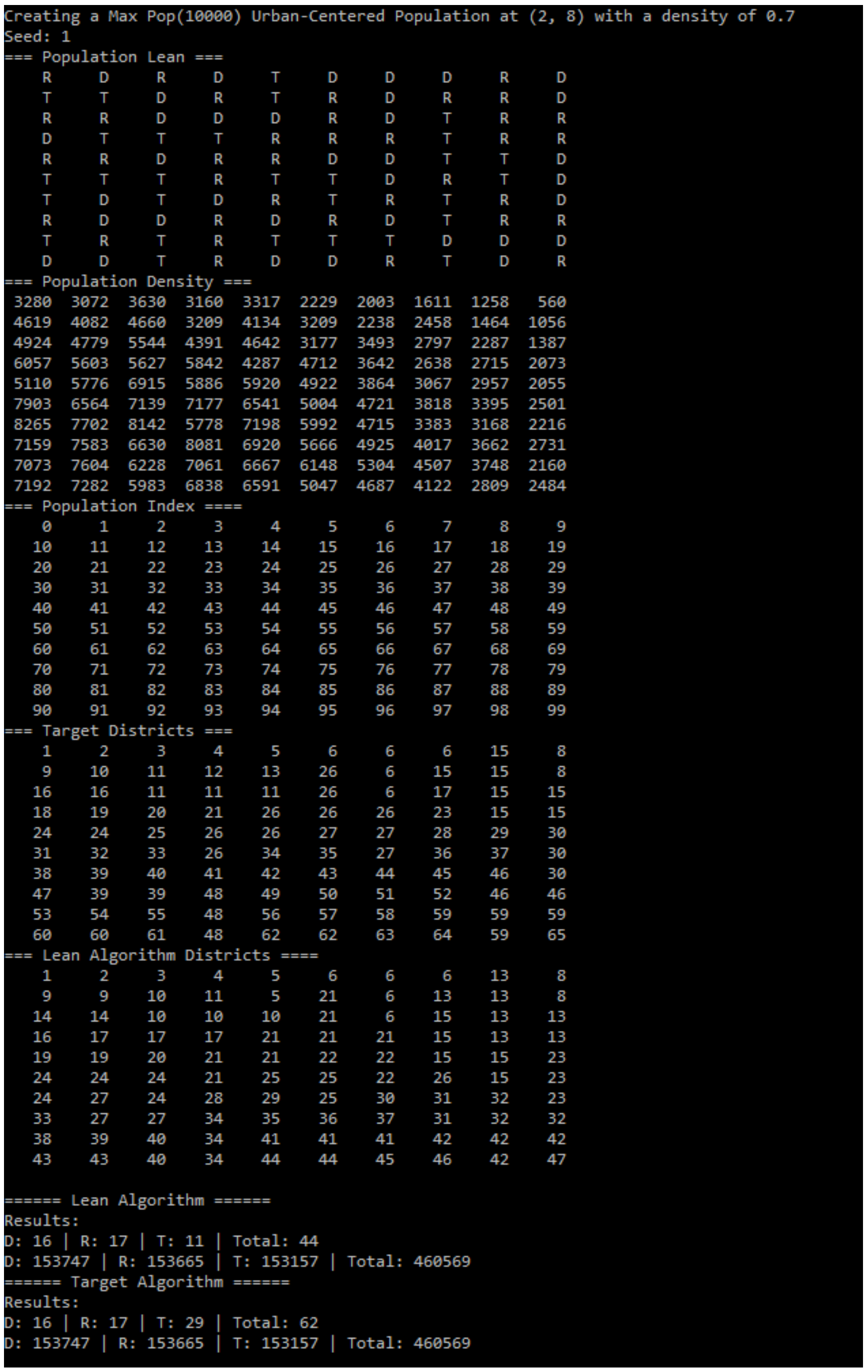
The project contains several printing functions that output data to the console and outputting functions that save this output to readable text files:

- **printPopulationLean()** - outputs the leans of each population cell in a formatted grid

- **printPopulationDensity()** - outputs the number of voters in of each population cell in a formatted grid
- **printPopulationIndex()** - outputs the index of each population cell in the population cell vector in a formatted grid
- **printDistrictIDGrid()** - outputs the ID tags associated with the district for each population cell, effectively showing the district map
- **printAlgorithmResults()** - counts the votes for each party both at the district level (representative) and voter level (popular)

Do note that this project is not designed to be an end-user program. We intend for this project to be used as a platform for developing more robust solutions to the issues and complexities surrounding gerrymandering. A basic implementation of our program could be similar to the following:

```
unsigned int num_rows = 10;
unsigned int num_cols = 10;
unsigned int max_population = 10000;
std::vector<std::string> parties = { "R", "D", "T" };

std::vector<PopulationCell> pop = genPopGridUrbanCenter(
    ↪ max_population, num_rows, num_cols, parties, 1);

Districting leanAlgorithm(pop, num_rows, num_cols);
Districting targetAlgorithm(pop, num_rows, num_cols, 1, "T");
std::cout << "===␣Population␣Lean␣===" << std::endl;
printPopulationLean(pop, num_rows, num_cols);
std::cout << "===␣Population␣Density␣===" << std::endl;
printPopulationDensity(pop, num_rows, num_cols);
std::cout << "===␣Target␣Districts␣===" << std::endl;
printDistrictIDGrid(targetAlgorithm.getDistricting(), num_rows,
    ↪ num_cols);
std::cout << "===␣Lean␣Algorithm␣Districts␣====" << std::endl;
printDistrictIDGrid(leanAlgorithm.getDistricting(), num_rows,
    ↪ num_cols);
std::cout << std::endl;

outputPopulationGridSize(pop, num_rows, num_cols);
leanAlgorithm.outputDistricting("lean");
targetAlgorithm.outputDistricting("target");

std::cout << "======␣Lean␣Algorithm␣======" << std::endl;
printAlgorithmResults(leanAlgorithm);

std::cout << "======␣Target␣Algorithm␣======" << std::endl;
printAlgorithmResults(targetAlgorithm);
```



```
Creating a Max Pop(10000) Urban-Centered Population at (2, 8) with a density of 0.7
Seed: 1
=== Population Lean ===
    R     D     R     D     T     D     D     D     R     D
    T     T     D     R     T     R     D     R     R     D
    R     R     D     D     D     R     D     T     R     R
    D     T     T     T     R     R     R     T     R     R
    R     R     D     R     R     D     D     T     T     D
    T     T     T     R     T     T     D     R     T     D
    T     D     T     D     R     T     R     T     R     D
    R     D     D     R     D     R     D     T     R     R
    T     R     T     R     T     T     T     D     D     D
    D     D     T     R     D     D     R     T     D     R
=== Population Density ===
 3280  3072  3630  3160  3317  2229  2003  1611  1258   560
 4619  4082  4660  3209  4134  3209  2238  2458  1464  1056
 4924  4779  5544  4391  4642  3177  3493  2797  2287  1387
 6057  5603  5627  5842  4287  4712  3642  2638  2715  2073
 5110  5776  6915  5886  5920  4922  3864  3067  2957  2055
 7903  6564  7139  7177  6541  5004  4721  3818  3395  2501
 8265  7702  8142  5778  7198  5992  4715  3383  3168  2216
 7159  7583  6630  8081  6920  5666  4925  4017  3662  2731
 7073  7604  6228  7061  6667  6148  5304  4507  3748  2160
 7192  7282  5983  6838  6591  5047  4687  4122  2809  2484
=== Population Index ====
    0     1     2     3     4     5     6     7     8     9
   10    11    12    13    14    15    16    17    18    19
   20    21    22    23    24    25    26    27    28    29
   30    31    32    33    34    35    36    37    38    39
   40    41    42    43    44    45    46    47    48    49
   50    51    52    53    54    55    56    57    58    59
   60    61    62    63    64    65    66    67    68    69
   70    71    72    73    74    75    76    77    78    79
   80    81    82    83    84    85    86    87    88    89
   90    91    92    93    94    95    96    97    98    99
=== Target Districts ===
    1     2     3     4     5     6     6     6    15     8
    9    10    11    12    13    26     6    15    15     8
   16    16    11    11    11    26     6    17    15    15
   18    19    20    21    26    26    26    23    15    15
   24    24    25    26    26    27    27    28    29    30
   31    32    33    26    34    35    27    36    37    30
   38    39    40    41    42    43    44    45    46    30
   47    39    39    48    49    50    51    52    46    46
   53    54    55    48    56    57    58    59    59    59
   60    60    61    48    62    62    63    64    59    65
=== Lean Algorithm Districts ====
    1     2     3     4     5     6     6     6    13     8
    9     9    10    11     5    21     6    13    13     8
   14    14    10    10    10    21     6    15    13    13
   16    17    17    17    21    21    21    15    13    13
   19    19    20    21    21    22    22    15    15    23
   24    24    24    21    25    25    22    26    15    23
   24    27    24    28    29    25    30    31    32    23
   33    27    27    34    35    36    37    31    32    32
   38    39    40    34    41    41    41    42    42    42
   43    43    40    34    44    44    45    46    42    47

====== Lean Algorithm ======
Results:
D: 16 | R: 17 | T: 11 | Total: 44
D: 153747 | R: 153665 | T: 153157 | Total: 460569
====== Target Algorithm ======
Results:
D: 16 | R: 17 | T: 29 | Total: 62
D: 153747 | R: 153665 | T: 153157 | Total: 460569
```

Figure 1: Console output from source code above

3 Code Overview

3.1 Voter Class

We used the Voter class to represent a person of voting status in the US; each voter has an

- **affiliation** that describes the party a voter sides with - used to calculate the overall lean of a district (see section 3.2)
- **ID** that is unique to every voter - used to keep track of each voter in a district (see section 3.2)

The Voter class has 2 methods:

- **getID()** - returns the ID of the voter
- **getAffiliationCode()** - returns the party code the voter affiliates with

The textbook called for other functions that were specific to a two-party analysis, but ultimately did not use them since the scope of our project includes a multi-party system with unique party codes.

3.2 District Class

We used the District class to represent a district of voters in the US; each district has a

- **vector of voters** - used to cycle through each voter in a district and tally up their affiliations for each party
- **ID** - used to keep track of each individual district in a group of districts (which is seen in the Districting class in section 3.4)
- **party code** and **party count** for the two parties with the most votes - The first party is the most popular party in the district while the second party is the second most popular party. Ideally, we would use the second party statistics to conduct runoff simulations since most US districts require a majority vote. To program this, we would have to add a ranked list of preferred affiliations for each voter and implement a rank-based system for conducting these runoffs (so that voters affiliated with the third parties are considered). This proved too complex for the time allotted.
- **map** of all parties with their respective counts - used to tally up the number of affiliations for every party. The first and second party codes and counts are then chosen from this.

The District class has 8 methods:

- **updateAffiliationCounts()** - updates the district's party affiliation map

- **getAffiliationCounts()** - returns the district's party affiliation map
- **updateStats()** - updates the district's statistics
- **getNumberOfVoters()** - returns the number of voters in the district
- **lean()** - returns the popular affiliation of the voters in the district
- **firstPartyCount()** - returns the number of voters affiliated with the popular party
- **secondPartyCount()** - returns the number of voters affiliated with the second most popular party
- **getVoters()** - returns the vector of voters in the district

3.3 PopulationCell Class

We used the PopulationCell class to represent an area of a larger population of voters in the US (you can think of a population cell as a neighborhood); each population cell has a

- **map** of all parties with their respective counts - used to tally up the number of voters affiliated with each party
- **lean** - used to describe the popular affiliation of the voters in the population cell
- **size** - describes how many voters are in the cell
- **vector of voters** - used to cycle through each voter in the cell and tally up the number of voters affiliated with each party
- **double linked list** of other adjacent population cells with the same lean - used in the algorithms that perform redistricting

The PopulationCell class has 7 methods:

- **getLean()** - returns the lean of the population cell
- **getCounts()** - returns the number of votes for each party
- **getSize()** - returns the number of voters in the cell
- **getVoters()** - returns the vector of voters in the cell
- **updateStats()** - updates the population cell's statistics
- **linkPopulationCell()** - adds population cell specified as a parameter to this population cell's linked list
- **changeUpstream()** - recursively works up the linked list, changing the indices to match the latest linkage

3.4 Districting Class

We used the Districting class to represent a collection of districts in a large area (you can think of a Districting object as the state of Texas with all of the districts in it); each Districting object has a

- **vector of districts** - contains all of the individual districts created from the specified population distribution and algorithm
- **map** of the index of a population cell to the index of a district - used to keep track of the population cells in each district
- **map** of all parties with their respective counts on the voter level - used to see how the entire population of a Districting object affiliates with each party
- **map** of all parties with their respective counts on the district level - used to see how each district in the Districting object affiliates with each party
- **integer** with the total number of voters - used to track the size of the entire population

The Districting class has 10 methods:

- **loadDistricts()** - creates districts out of the population cells
- **countVotes()** - counts the number of district and popular votes for each party
- **getDistricts()** - returns a vector of all the districts in the Districting object
- **getDistricting()** - returns a vector with the indices of the population cells' corresponding districts found in **getDistricts()**
- **outputDistricting()** - outputs the districting vector to a text file
- **getPartyCounts()** - returns the map of the number of districts affiliated with each party
- **getPopularCounts()** - returns the map of the number of voters affiliated with each party in the entire districting
- **getNumOfVoters()** - returns the number of voters in the Districting object
- **districtLean()** - runs the ”lean” algorithm (refer to section 5.1)
- **districtTarget()** - runs the ”targeting” algorithm (refer to section 5.2)

4 Population Cell Algorithms

We created two algorithms for generating population distributions. These functions should serve as the foundation for developing smarter algorithms to simulate more realistic population distributions.

4.1 Uniform Random

The first algorithm is the simplest and least realistic.

```
std::vector<PopulationCell> genPopGridUniformRandom(unsigned int
    ↪ max_population, unsigned int num_rows, unsigned int
    ↪ num_cols, std::vector<std::string> parties, unsigned int
    ↪ seed) {
        std::vector<PopulationCell> grid;

        int latest_ID = 0;

        std::srand(seed);

    for (unsigned int row = 0; row < num_rows; row++) {
            for (unsigned int col = 0; col < num_cols; col++) {
                        PopulationCell cell(std::rand() %
                            ↪ max_population, parties, latest_ID);
                        grid.push_back(cell);
                }
        }
        return grid;
}
```

Analyzing the function parameters, this algorithm builds a population cell array by using the following:

- **max population** - maximum number of voters in a district
- **num rows** - number of rows in the two dimensional population cell grid
- **num cols** - number of columns in the two dimensional population cell grid
- **parties** - party distribution to randomly pick party affiliations from (to increase odds of one party over another, add more copies of that party's code to the vector)
- **seed** - integer that seeds the random number generator

First, we create a vector to store the population cells in and an integer to keep track of the next available voter ID. The function **srand()** is called to seed the random number generator.

```
std::vector<PopulationCell> grid;

int latest_ID = 0;

std::srand(seed);
```

We then iterate through each column of each row of the grid we want to create.

```
for (unsigned int row = 0; row < num_rows; col++) {
        for (unsigned int col = 0; col < num_cols; row++) {
```

At each position in the grid, a cell is created with a random amount of voters below the maximum, each with an affiliation from the party distribution and a unique ID. See section 3.3 for more information on how these population cells are created.

```
PopulationCell cell(std::rand() % max_population, parties,
    ↪ latest_ID);
        grid.push_back(cell);
```

4.2 Urban Center

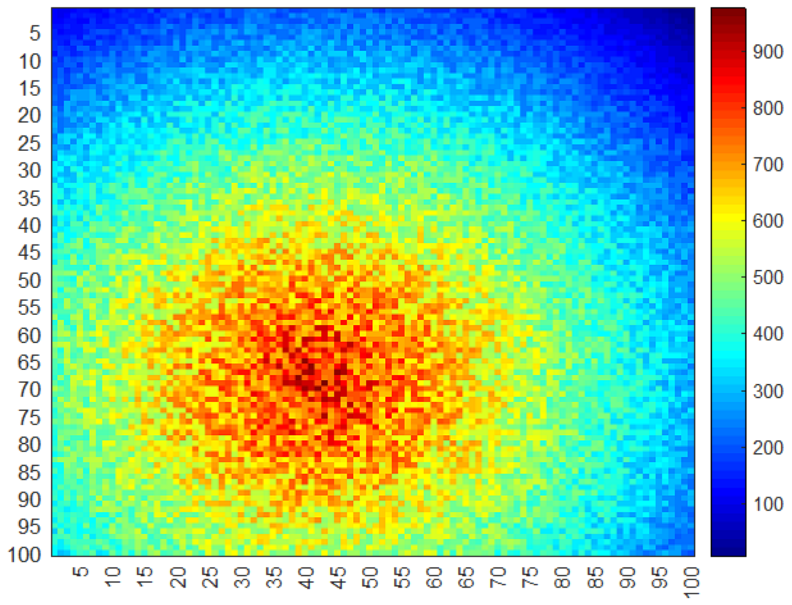


Figure 2: Population distribution produced by ”Urban Center”

While the first algorithm is simple, it is very unrealistic. Most population distributions are not uniformly random, but instead focused around cities. This algorithm picks a random position within the grid to designate as the ”urban center” and then populates the rest of the grid around it to mimic the distribution of major cities.

```
std::vector<PopulationCell> genPopGridUrbanCenter(unsigned int
    ↪ max_population, unsigned int num_rows, unsigned int
    ↪ num_cols, std::vector<std::string> parties, double density
    ↪ , unsigned int seed) {
        std::vector<PopulationCell> grid;
        int latest_ID = 0;
        std::srand(seed);

        int center_x = rand() % num_cols;
        int center_y = rand() % num_rows;
        double max_dx = double (num_cols) - (center_x);
        double max_dy = double (num_rows) - (center_y);
        if (max_dx < center_x) {
                max_dx = center_x;
        }
        if (max_dy < center_y) {
                max_dy = center_y;
        }

        for (unsigned int y = 0; y < num_rows; y++) {
                for (unsigned int x = 0; x < num_cols; x++) {
                        double dx = std::abs(center_x - int(x));
                        double dy = std::abs(center_y - int(y));
                        double cell_pop_percentage = 1 - (std::pow((
                            ↪ dx*dx + dy*dy), 0.5) / (std::pow((
                            ↪ max_dx* max_dx + max_dy* max_dy + 1),
                            ↪  0.5)));
                        double max_cell_pop = cell_pop_percentage *
                            ↪ max_population;
                        double min_cell_pop = max_cell_pop * density
                            ↪ ;
                        unsigned int size = std::rand() % int(
                            ↪ max_cell_pop - min_cell_pop) + int(
                            ↪ min_cell_pop);
                        PopulationCell cell(size, parties, latest_ID
                            ↪ );
                        grid.push_back(cell);
                }
        }
        return grid;
```

This function's parameters are similar to the previous one's except with the addition of **density** which specifies how dense the population should be as it moves further from the urban center.

First, we randomly select a position for the urban center.

```
int center_x = rand() % num_cols;
int center_y = rand() % num_rows;
```

Then, for every cell, we use the following formulas to determine the range for how many voters it should contain:

Max Cell Pop. $= (1 - \sqrt{\frac{dx_*^2+dy_*^2}{dx_m^2+dy_m^2+1}})maxpop$

Min Cell Pop. $=$ (Max Cell Pop.) * density

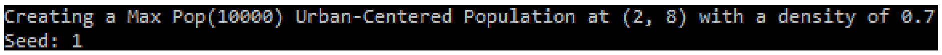
```
double cell_pop_percentage = 1 - (std::pow((dx*dx + dy*dy), 0.5)
    ↪ / (std::pow((max_dx* max_dx + max_dy* max_dy + 1), 0.5)));
double max_cell_pop = cell_pop_percentage * max_population;
double min_cell_pop = max_cell_pop * density;
```

5 Districting Algorithms

These algorithms follow a basic set of assumptions when drawing the districts.

- **Continuous** - all districts must be a continuous body with no breaks or separations between the population cells.
- **No Majority** - no districts require a majority to have a designated party affiliation.
- **No Voter Apathy** - all voters in every population cell will participate and any voter apathy will not be considered in designating the district party affiliation.

For the demonstrations, we will use the following population distribution:



```
Creating a Max Pop(10000) Urban-Centered Population at (2, 8) with a density of 0.7
Seed: 1
```

5.1 Lean

This algorithm seeks to group similar voters together regionally based on lean. If any cells are next to each other and have the same lean, they will be merged into a single district.

First, we iterate through each position in the population grid from left to right, top to bottom. Since the data is stored in a one dimensional vector, we need to calculate the index.

```
for (unsigned int y = 0; y < pop_num_rows; y++) {
        for (unsigned int x = 0; x < pop_num_cols; x++) {
                unsigned int index = y * pop_num_cols + x;
```

For each cell, we need to check the top neighbor then the left neighbor. If the cell doesn't link to either, then it must be its own district. First, we check the top neighbor if possible. If these cells have the same lean, then link them and indicate that no new district is needed.

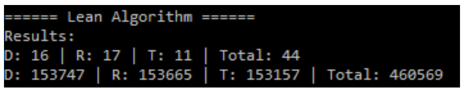
```
if (y > 0) {
        unsigned int top_index = ((y - 1) * pop_num_cols) + x;
        if (this_lean == population[top_index].getLean()) {
                population[index].linkPopulationCell(&population[
                    ↪ top_index]);

                new_district = false;
        }
}
```

Next, we check the left neighbor. An additional check is needed to insure that these districts aren't already linked or we will create a circular linked-list.

```
if (x > 0) {
        unsigned int left_index = (y * pop_num_cols) + (x - 1);
        if (population[left_index].getLean() == this_lean &&
            ↪ population[index].district_index != population[
            ↪ left_index].district_index) {
                population[index].linkPopulationCell(&population[
                    ↪ left_index]);

                new_district = false;
        }
}
```

Lastly, if neither of these cases succeeded in linking the cell, this cell must be a new district.

```
if (new_district) {
        population[index].district_index = next_district_index;
        next_district_index++;
}
```

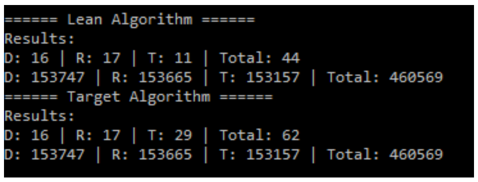


```
====== Lean Algorithm ======
Results:
D: 16 | R: 17 | T: 11 | Total: 44
D: 153747 | R: 153665 | T: 153157 | Total: 460569
```

Figure 3: Districting results produced by the Lean Algorithm

5.2 Targeting

Unlike the lean algorithm, the targeting algorithm actively tries to increase a particular party's district representation. In fact, it produces the most districts possible for the targeted party and the least possible districts for the other parties. It accomplishes this effect by simply executing the lean algorithm, but only linking cells that don't affiliate with the targeted party. The result is individual districts for every single population cell affiliated with the targeted party.



```
====== Lean Algorithm ======
Results:
D: 16 | R: 17 | T: 11 | Total: 44
D: 153747 | R: 153665 | T: 153157 | Total: 460569
====== Target Algorithm ======
Results:
D: 16 | R: 17 | T: 29 | Total: 62
D: 153747 | R: 153665 | T: 153157 | Total: 460569
```

Figure 4: Districting results produced by the Target and Lean Algorithm

To do this, all we need to do is consider this condition when linking two cells.

```
this_lean != target_party
```

Implemented in both checks for the top and left neighbors:

```
if (y > 0) {
        unsigned int top_index = ((y - 1) * pop_num_cols) + x;
        if (this_lean != target_party && this_lean == population[
            ↪ top_index].getLean()) {
                population[index].linkPopulationCell(&population[
                    ↪ top_index]);
                new_district = false;
        }
}
if (x > 0) {
        unsigned int left_index = (y * pop_num_cols) + (x - 1);
        if (this_lean != target_party && population[left_index].
            ↪ getLean() == this_lean && population[index].
            ↪ district_index != population[left_index].
            ↪ district_index) {
                population[index].linkPopulationCell(&population[
                    ↪ left_index]);
                new_district = false;
        }
}
```

Looking at the figures 3 and 4, we can see how the two algorithms differ in their nature. In the second figure, you can see where previously organized districts are now fragmented into their own individual districts. This is because targeted affiliations will not be organized into districts in order to increase the representation of that affiliation.

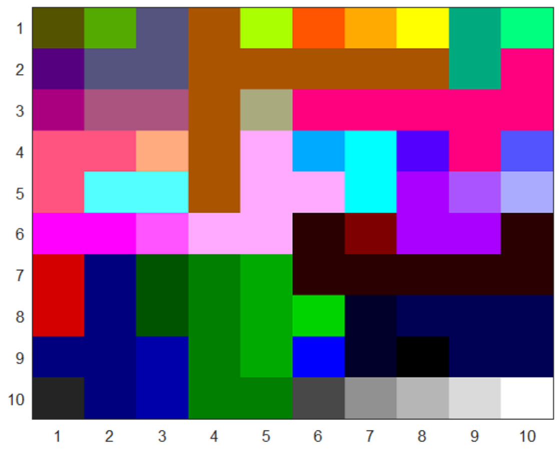


Figure 5: Districting produced by the Lean Algorithm

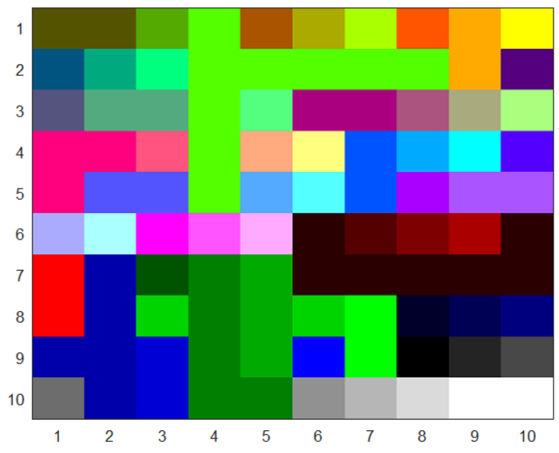


Figure 6: Districting produced by the Target Algorithm

6 Simplifications and Limitations

This project is by no means a realistic model for redistricting an actual region or state. The goal of this project was to lay the foundation for a comprehensive model that could objectively draw districts and thus solve the issue of partisan gerrymandering. Since our development was on a time constraint, we had to simplify the problem greatly and accept some limitations for our current model.

In addition to the assumptions discussed in the beginning of section 5 for drawing districts, we also have to consider the multitude of variables associated with representation of an electorate. The geography of a region can also affect the mapping of districts. The algorithms we wrote do not consider regional enclaves of any type (cultural, socioeconomic, etc). This results in a map of districts that splits neighborhoods and separates population areas that shouldn't be separated. For example, if a single neighborhood wants their roads renewed, that opinion should not be split between two districts that may result in that opinion not being heard by the representatives of those districts.

7 Challenges

The biggest challenge of the project was coming up with the algorithms we used to implement redistricting; it took hours of brainstorming and debugging.

At first, we thought we would attempt to group population cells together by going through each cell and comparing it to the neighbor on its right, and, once that was done, going through each cell and comparing it to the neighbor below it. Even if we looped through that process, nothing would change after the first alterations.

Then, we decided to improve that process by adding the functionality of going right to left and bottom to top. This actually worked with the example we were testing (a 3x3 grid of population cells). Unfortunately, our strategy no longer worked with larger grids. Instead of grouping all the similarly-affiliated cells, there would be adjacent cells with the same affiliations that were not in the same district.

We were able to solve this problem with loops; we decided to start counting the number of changes each method (left to right, top to bottom, right to left, and bottom to top) would make on our grid and we figured that if there were no changes to our grid after a series of 20 iterations, we would have successfully redistricted. This strategy worked in most cases. However, in some cases (especially with larger grids) there would be a pattern of changes each time. For example, every iteration there would be two changes, so it would never get solved. We realized the main problem with this approach was we kept iterating in the same pattern, so if we did get stuck, it was impossible to get unstuck.

Our solution was random numbers: We made a number that would randomly choose one of the four grid-changing methods discussed earlier. That way if we ever hit a pattern, it was possible for us to get out of it. This ended up working for all the grids we gave it. The downside to this algorithm was that it took too long depending on the random number generation. Furthermore, if it repeated the same method twice (ex: right to left and then right to left again), the grid wouldn't change, so the iteration would have been wasted. Although this algorithm worked, we decided we wanted one that was faster and more reliable, which led us to the linked-list system we have implemented now.