Division of Labor

Sasha Bakker

- Wrote code to read data from a text file to a 2d integer array
- Wrote the borda count method
- Wrote the instant runoff method

Files:
voting.c
borda.h
instant_runoff.h

Text file functions:
createMatrix()
transform_data()
printMatrix()
freeMemory()

Borda count functions:
borda_count_method()
find_borda_counts()
find_maximum_of_array()
has_multiple_maxima()
find_minimum_of_array()
find_borda_winner()

Instant runoff functions:
find_arr_min()
find_winner()
instant_runoff_method()

Shoale Badr

- Wrote code to read/deal with command input
- Wrote the schulze beatpath method
- Wrote the pairwise comparison method

Files:
voting.c
schulze_beatpath.h
pairwise_comparison.h

Schulze beatpath functions:
findMax()
findMin()
shulze_beatpath()

Pairwise comparison functions:
pairwise_comparison()

Overall Approach

Reading in the text file:

> File opened in read mode via `fopen()`

> First line of text file is red via `fscanf()` and stores the values of the number of rows and columns at the addresses of integers `rows` and `cols`

> Memory is allocated for the data to be stored in a 2D integer array with dimensions `rows` and `cols`

> Each line of the text file is read and individual numbers are stored as elements in the 2D array `data`

> The file is closed via `fclose()`

Representing the ballots:

> The 2D array `data` in `main()` represents the ballots as read exactly from the text file where rows = ballots, cols = candidates, elements = ranking (1st, 2nd, …)

> This array `data` is used for Method Pair 2. It is transformed for Method Pair 1 via the function `transform_data()` such that cols = ranking and elements = candidates.

> The memory of the arrays are freed after use via `free()`

Overall code structure:

- All voting calculations were done function-wise and then placed in separate header (".h") files for easy readability of the main() function
- Command line arguments were implemented in the main() function and then passed into each of the voting method functions
- All parts of the code are well-documented and have clear variable names

Method Pair 1

Borda Count

```
// create matrix of borda points candidates can receive
// while more than one candidate has the maximum borda count:
    > Get candidate borda counts
    > Find maximum borda counts
    > If more than one candidate has the maximum,
        "remove" candidates with the minimum count from the ballot by setting the points they can receive to zero
    > If there are no more candidates left, all win!

// If one candidate has the maximum, they win!
```

Instant Runoff

```
// create matrix of initial scores
// while zero candidates or more than one candidate holds the majority (>50%):
    > Set candidate score (number of first place votes) to the initial score (0 if participating, -1 if not participating)
    > Update the participating candidate scores
    > Compute number of candidates that hold the majority
    > If more than one candidate holds majority, "remove"
        candidates with the minimum % from tha ballot by setting their initial score to -1

// If one candidate holds majority, they win!
```

- The approaches work well because they produce the correct results for each text file. However, I do not believe they are the most efficient because I misunderstood the data files and had to transform the data matrix via `transform_data()` in `main()` before executing these, such that the rows = ballots, cols = ranking (1st, 2nd,...), elements = candidate.
- These methods would be improved by computing the results according to the original data matrix. Another improvement would be to combine the maximum/minimum-finding components/functions that these methods share.

Method Pair 2

Pairwise Comparison

```
// create matrix that compares each candidate’s popularity
// if the comparison matrix entry [i][j] < entry [j][i]:
        > Add one point in the corresponding entry in the points array
        > Find the candidate with the maximum amount of points
        > If more than one candidate has the maximum amount of points, they tie

// If one candidate has the maximum, they win!
```

Schulze Beatpath Method

```
// create matrix that compares each candidate’s popularity
// use the Floyd-Warshall algorithm to create a matrix of “strongest paths” - this essentially fills each entry with the maximum between an entry in the comparison matrix and the minimum between another entry (same row) and its mirror
// use the same “points” method as in pairwise comparison to fill a “points” array
        > Find the candidate with the maximum amount of points
        > If more than one candidate has the maximum amount of points, they tie

// If one candidate has the maximum, they win!
```

- These approaches work well because they produce the correct results for each text file. They could, however, be improved by creating a separate function to create a comparison matrix, as this is shared by both methods. This was unable to be accomplished due to the complexity of passing a 2D array through several functions.