On the Creation of an Electoral Simulator

Riggs Markham

Spring 2021

Abstract

This work describes the construction of a web application that simulates electoral outcomes given polling data as an input. The aim of this application is to create a user-friendly interface that allows a user to quickly and easily input a poll and then see the projected outcome of this election under various electoral systems. Additionally, we created algorithms to evaluate the winning choices of those various electoral systems: first past the post, ranked choice voting, top-two runoffs, approval voting, and Copeland's method. The web application, after simulating the outcome of an election, shows the resulting proportions that each choice in an election won in both plots and tables.

1 Introduction

1.1 Motivation

We consider the problem of creating a user-friendly web application to simulate an election using a variety of electoral systems. Solutions to this problem have applications in the realm of public policy, where political observers prefer to see fast, secure, and most importantly, accurate, election counts in order to verify the results of an election. Electoral systems that are quick to compute also take precedence in more informal electoral settings, like private clubs, or in just making a consensus decision among any group of people. Additionally, one of the principle arguments against electoral reform is the supposed difficulty of both evaluating the results and casting the votes in alternate systems, so proper analysis of algorithmic complexity is required for any valid analysis of voting systems. Solutions to this problem have applications in public policy, political media, electoral campaigns, and for any organization that uses an electoral process to make decisions. Simulating elections allows organizations to choose electoral systems that best fit their needs and allows observers to prepare for expected electoral outcomes.

1.2 Background

In an election, voters have preferences about candidates, parties, or choices that they express by voting in the electoral system. For each election, there is a list of n choices, which could represent candidates, parties, or some other kind of choice, $c = \{c_1, c_2, ..., c_n\}$, where each c_i represents an election choice. Each voter has a *preference profile*, the record of that voter's preferences, which contains a list of integers of length n $\{p_1, p_2, ..., p_n\}$, with each integer p_i corresponding to an election choice c_i.

Each integer p_i represents a value that a voter places on an election choice. What this integer specifically refers to can be arbitrarily defined, but in this application, it will represent a score that a voter assigns to a choice, with higher numbers representing a more preferable choice and lower numbers representing a less preferable choice. This includes both positive and negative numbers and has no set bounds. Negative numbers refer to a choice that the voter does not approve of or one that the voter dislikes, but a voter still might prefer that choice to others. This *preference profile* system allows for a voter's preferences to be input into different electoral algorithms without modification.

1.3 Problem Definition

Let there be a set of preference profiles for an electorate $p = \{p_1, p_2, ..., p_n\}$ and a set of corresponding choices $c = \{c_1, c_2, ..., c_n\}$ for an election with n choices. Let there be a specific electoral system with an algorithm $f(x, y) = a$, where a is the winning choice of the election given a set of preference profiles x with a set of choices y.

Find an algorithm $f(p, c) = a$, such that a is the winner of an election in the following electoral systems: an arbitrary positional scoring rule, instant runoff voting, and various proportional representation.

2 Related Work

2.1 Computational Social Choice

The mathematical field of social choice theory concerns methods of collective decision making [1]. In computational social choice theory however, principles of computer science and artificial intelligence are applied to the problems of social choice theory. Historically, much of the work in social choice theory is highly theoretical, however, computational and concrete design and analysis greatly enhances the field.

The other major realm of computational social choice is the application of the principles of social choice theory to issues of computer science and artificial intelligence. Areas of study within the field of computational social choice include preference aggregation, voting theory, fair resource allocation, and coalition formation [1]. A specific problem of this field is the winner prediction problem in elections, a favorite of pundits, campaigns, and theorists.

Formally, this problem is deemed the (ϵ, δ)-*winner determination problem*. Given an election with n voters, in which the margin of victory is at least ϵn votes, the goal is to determine the winner with a probability of at least $1 - \delta$ [2]. Upper and lower bounds for the number of samples needed to solve this problem for many common voting rules were found by Bhattacharyya and Dey [3]. Since there is a large number of distinct voting systems of different complexities, Mogos and Mogos [4] proposed a formal representation of voting methods using complexity functions and proved theorems related to the comparison of several, single-winner voting methods.

Another problem in this field is the computation of the margin of victory. In some electoral systems, such as in plurality voting, computing the margin of victory is simple, but in some, such as instant runoff voting, this computation is rather difficult [5]. Xia [6] investigated the computational complexity of computing the margin of victory for various voting rules, including approval voting and positional scoring rules.

2.2 Positional Scoring Rules

A *positional scoring rule* is a rule by which a score, designated by a number, is assigned to each candidate for each individual's preference profile. The winning candidate under a positional scoring rule is the candidate is the candidate with the maximum sum of scores [1].

The dominant voting method in the United States, *plurality voting* or *first-past-the-post*, operates under a rule that gives a score 1 to the highest ranked candidate and gives a score of 0 to each other candidate. Another voting method, the *Borda count*, assigns scores from n, where n is the number of candidates, down to 1 for each candidate according to each voter's preference profile [1].

One way to model a positional scoring rule is by using a vector of weights, $(w_1, w_2, ..., w_n)$, that are assigns to candidates based on how a voter positions them on a ballot. By setting the top-ranked candidate's weight to 1 - the maximum weight - and the bottom ranked candidate's weight to 0 - the minimum weight, a vector $(1, w_2, w_2, ..., w_{n-1}, 0)$ is used [7]. This technique provides for a quick method of modeling different positional scoring rules. For example, plurality voting is represented by a vector $(1, 0, 0, ..., 0)$ and the Borda count, for an election with n candidates, is represented by $(1, \frac{n-2}{n-1}, \frac{n-3}{n-1}, ..., \frac{1}{n-1}, 0)$ [7].

2.3 Instant Runoff Voting

Instant runoff voting, also known as ranked choice voting or alternative vote, is a ranked preferential voting system that is used to elect one candidate in elections with more than two candidates [5]. This method is used in national election in Australia and Ireland and is used in many localities around the world. It is useful in eliminating the *spoiler effect*, a phenomenon in plurality voting where similar candidate split the votes of their supporters, allowing another candidate to win, even if most of the population prefers another to them [5].

As noted earlier, in plurality voting the margin of victory is simple to calculate and the runner-up candidate is easy to identify. In instant runoff voting identifying these can be difficult, i.e., the exact computation of instant runoff electoral margins is NP-Hard [5]. These metrics are useful in the case of risk-limiting audits, which rely on knowledge of the margin of victory and on random sampling in order to make sure that the election result was valid [8].

In calculating the margin of victory in an instant runoff election, two upper bounds and two upper bounds were given by Cary [8] to create an estimate with a time complexity that does not exceed $\mathcal{O}(n^2 \log n)$, where n is the number of candidates. Concurrently, Blom et al. [5] found an exact instant runoff margin computation method that, although it is exponential is the worst case, it runs efficiently in practice in real world examples.

2.4 Proportional Representation

Proportional representation is a class of electoral methods for a body of representatives that are elected in proportion to the preferences of voters. Proportional representation is an extremely common voting system; it is used in many nations across the world to elect legislatures, both national and subnational [9].

There are various methods and formulas with which proportional representative systems are implemented, but in its most common form, each voter votes for a party and each party gets to appoint representatives in a number of seats proportional to the number of votes that the party received [9]. Oftentimes, in an effort to make political blocs more coherent, parties are required to receive a certain number of votes, usually 3%-5% of the total votes, in order to have any representatives at all. As simply dividing the number of seats by the number of votes would result in fractional amounts of seats for certain parties, other methods are used to decide on the exact numbers of seats, so other methods are used. The two widely used classes of methods are known as *highest averages methods*, which contain the D'Hondt, Sainte-Laguë, and Huntington-Hill methods, and *largest remainder methods*, which contain the Hare and Droop quotas [9].

However, much of the literature surrounding proportional representation systems deals with more obscure and less frequently used rules, such as the *Chamberlin-Courant rule*, a rule which combines the Borda rule and proportional representation to select the most representative committee from a group of candidates [10]. Dey et al. [11] found efficient algorithms for elections where votes come one at a time, in a streaming fashion, to approximately identify a winning committee under several of these multi-winner proportional representation voting rules.

3 Description of the Application

The primary function of the application will be a simulator that takes input in the form of polling, uses that input polling to simulate the outcome of an election, and then displays a summary of the results of that simulation to the user. The polling data can be input in various forms and the elections are simulated using a variety of different electoral systems. This application will be hosted on a public website, riggsmarkham.com, and will allow users to input their own data and then see the results of the simulation.

3.1 Application Process Flow

The basic functioning of this web application consists of the processes described below and summarized in figure 1. To use the web application, a user will first navigate to the simulator from the homepage and then

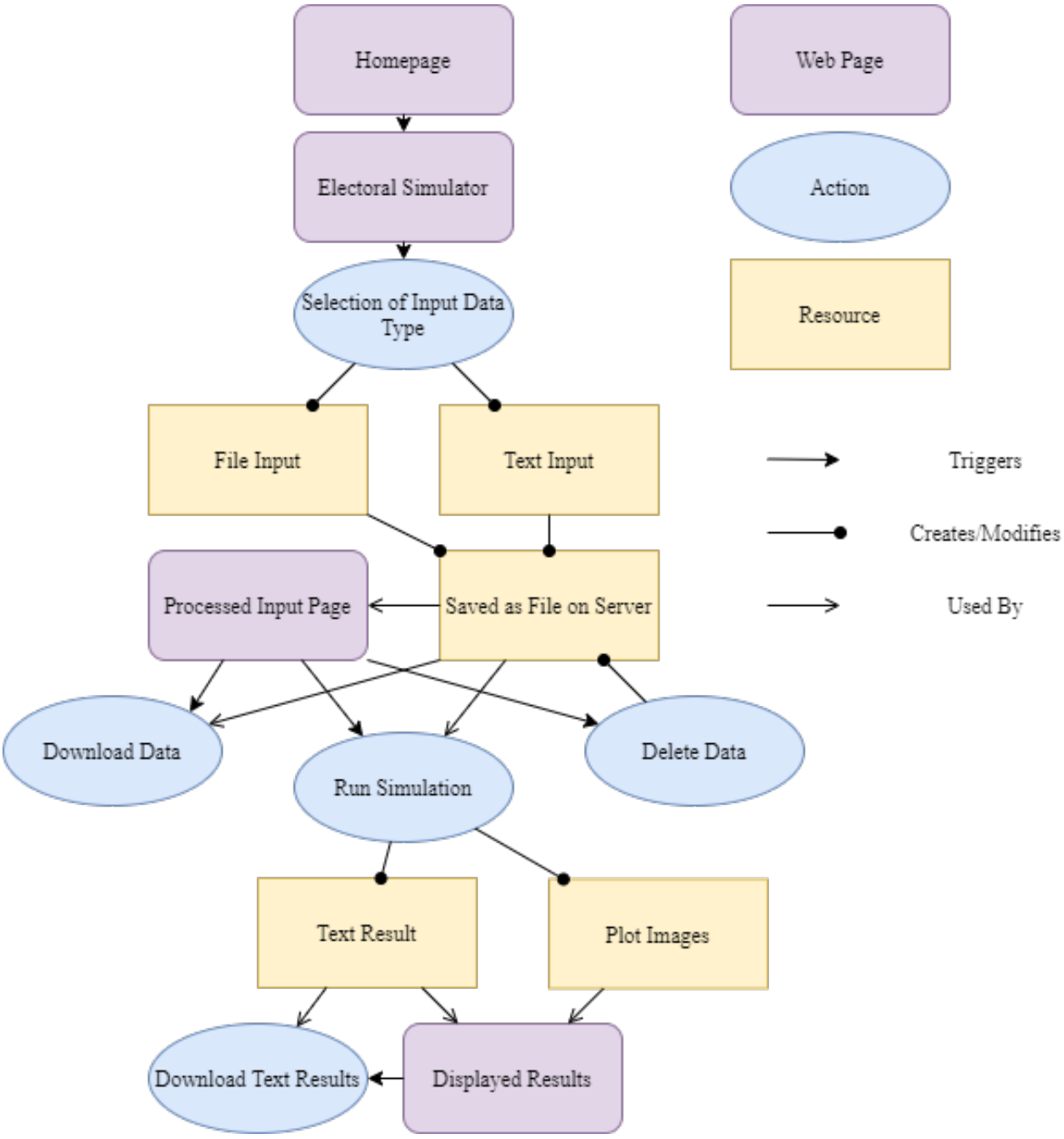


Figure 1: Flowchart of Basic Processes of Election Simulator

start using the buttons displayed on that page to select the format of the input data that they would like to use.

Then, they will have either selected to input data via either a file or by inputting text into forms on the webpage, and once they have submitted the data via one of these methods, it will be saved onto the server in a file format. The varieties of polling data that the user must input is described in section 3.3. The format and structure of this polling data - in its file form - is described in section 4.3. If the user uploaded a file, that file will be directly uploaded to the server. The format in which this If the user input data manually, then the application will use that data to create a file on the server containing that data.

Once a file is on the server, the server will give a summary of the entered data to the user with three options: download, run, and delete. *Download* will download the input file for the user - in this way, a user can input the same data that they input manually by using a file in the future, which is a significantly faster process. *Delete* will delete the file on the server. Finally, *run* will use the uploaded data file and run it through the simulation.

The simulation will calculate the projected results of the elections described by the input data. These results will be calculated based on the algorithms described in section 4.4 for the electoral systems described in section 3.2. From this simulation, plots of the results are created and then subsequently saved to the server, and, additionally, a textual description of the results is constructed and sent as a response to the webpage.

Then the webpage will display both the text result and all of the images of the plots. On this final page displaying their results, there will also be an option to download this textual result of the outcome of the elections.

3.2 Electoral Systems

The electoral systems that are simulated with this application are:

- First Past The Post
- Ranked Choice Voting
- Top-Two Runoff
- Approval Voting
- Copeland's Method

3.2.1 First Past the Post

First past the post (FPTP) electoral systems (also known as plurality voting or more formally as single-member plurality voting) simply allow each voter a single vote and the choice with the most votes wins. This simple voting system is widely used across the world, but is also heavily criticized by electoral systems experts for some of its perceived failings. Using the model used earlier to describe positional scoring rules, FPTP is a kind of positional scoring rule with the vector $(1, 0, 0, ..., 0)$, where one's first choice gets a full vote and everyone else gets nothing.

3.2.2 Ranked Choice Voting

Ranked choice voting (RCV) is an electoral system where voters are allowed to rank candidates in the order that they prefer them. In most implementations, voters are allowed to rank only their top n choices, instead of being able to rank all of the choices for the election. This is for both the convenience of voters, who will likely not have strong opinions on all of the choices, and the ease of the election administrators, not having to print complicated ballots or go through an extended counting process.

This sort of election is evaluated by ordering all of the choices by the number of 1st place votes that each choice received. Then, the choice with the lowest number of 1st place votes is eliminated, with each of the

votes that were previously going to it instead going to the voters' 2nd choice. This continues with the last place choice getting eliminated and their votes getting redistributed to the voters' next extant choices until one choice has more than half of the remaining votes [1].

This process can be made more efficient by, instead of simply eliminating the choice with the least votes, one eliminates all the choices for which their sum is less than the next highest ranked choice.

More formally, given an array $P = (x_1, x_2, ..., x_k)$ where each x_i represents the currently tabulated support for each electoral choice i of k possible choices and $x_i \geq x_{i+1}$ holds for all $i < k$; i.e. this is the decreasing ordered array of each choice's support. The set of choices that should should eliminated simultaneously is described by formula 1.

$$S = \Big(x_i \Big| \Big(\sum_{n=0}^{k-i} x_{k-n}\Big) < x_{i-1}\Big) \tag{1}$$

This simplification works because if the sum of S is less than the support of the lowest ranked choice in P outside of S, then there is no possibility that any member of S could receive enough redistributed votes to ever rank above that lowest ranked choice in P outside of S. Therefore, if evaluated using the normal method, every member of S will eventually be the worst ranked member of P, getting eliminated. This makes the process more efficient by removing the need for redistributing votes to other members of S that will soon be eliminated. Instead, votes are simply redistributed to the highest ranked candidate on each voter's list that is outside of S.

3.2.3 Top-Two Runoff

A Top-Two runoff is an electoral system that is usually implemented in a two-stage process. Initially, all choices are allowed in an election, but after the first round, all choices but the top two vote recipients are eliminated. Then, at some point later, a second round of voting is held, but this time, only those two choices are allowed. The winner of that round is the winner of the overall election. This method is used in many national single-office elections around the world, with presidents specifically often being elected by this process. Notably, French and Brazilian presidential elections both use this process. This system is similar to RCV, except two rounds are usually held instead of one, and there is only one round of eliminations instead of up to $n - 2$ rounds for n choices. This electoral system eliminates the sort of spoiler effect that dooms third parties in American presidential elections, but it only has the risk of causing major political factions to be excluded because they included too many candidates, diluting their votes and allowing two members of the opponent's faction to advance instead. These sorts of elections are also sometimes known as jungle primaries or open primaries in the United States when they are used as primary elections - mainly in Louisiana, California, and Washington.

3.2.4 Approval Voting

Approval voting is an electoral system that allows each voter to vote for as many choices as they would like to vote for (or as few), and the choice that has the most votes is declared to be the winner. This system is very simple and is held in high regard by election experts, but it is relatively rare in terms of real-world use.

3.2.5 Copeland's Method

Copeland's method is a variety of Condorcet electoral systems that produce a Condorcet winner when there is one. A Condorcet winner is a choice in an election that would win in a two-choice election against each of the other choices in a plurality election. In other words, the Condorcet winner is the choice in an election that voters prefer to every other choice. There are many method's that fulfill this *Condorcet winner criterion*, which means that a system always chooses the Condorcet winner if it exists, but there are some cases where it does not exist and this is where these alternate methods differ. Copeland's method specifically gives the

[1] As voters in most implementations are technically not *required* to rank as many choices as they can, oftentimes, every choice on a voter's ballot can be eliminated, at which point their votes are usually discarded

choice who wins the most one-on-one matchups as the winner. Since if a choice wins its matchup with every other choice, it will trivially have the highest number of wins in one-on-one matchups, this method will always choose the Condorcet winner.

3.3 Varieties of Polling

This simulation will allow users to input four different varieties of polls with which simulations will be run. I will refer to these different polling formats as follows:

- Single Choice
- Ranked
- Approval
- Pairwise

3.3.1 Single Choice

Single choice polls simply consist of each respondent's first choice of the candidate, party, or choice that they would like to vote for and provide no additional information. They are the most traditional and by far most the common format for election polling. This form of polling provides adequate information for first-past-the-post elections and other sorts of elections that only allow voting for a single option, but for other sorts of single winner electoral systems, they do not contain enough information to be useful beyond discerning the dominant first choices of voters.

3.3.2 Ranked

Ranked polls consist of recording a voter's ranked preferences from their 1st choice to their nth choice, with n being a number less than or equal to the number of choices available to the voter. While uncommon, these kinds of polls are found occasionally, particularly in the form of pollsters asking for both a voter's first and second choices for an election. While this sort of polling is often done for elections where voters can vote for a single choice - often in pursuit of trying to more accurately predict the final outcome by assessing how many voters could switch their votes to their second favorite choices - this sort of polling is essential for obtaining a clear picture of the electorate for an election with a ranked electoral system, such as ranked choice voting or the Borda count. When done with a high n - particularly if every possible choice is ranked by respondents - these sorts of polls have an extraordinarily large amount of information, and with it, it is reasonable to simulate many different kinds of elections: from first-past-the-post elections to Condorcet methods.

3.3.3 Approval

Approval polls consist of a recording of the proportion of voters that approve of each possible choice presented in an election. In American electoral polling, these polls are often performed alongside traditional single choice polls, but, as they are not directly asking the question that is usually at stake in American elections - Will you vote for Candidate A or Candidate B? - they are not used as heavily in predicting electoral outcomes as single choice polls. Approval polling is also extremely common in politics outside of specific electoral scenarios. One of the most common measures of the *success* of an American president is their job approval rating, measured using repeated polls on whether respondents approve of the work of that president [2]. However, for elections that use the approval voting method, this sort of polling is essential; since voters can and often do approve of more choices than one (or none of the choices), approval polling gives information that cannot easily be surmised from the outcome of a single choice poll.

[2]More specifically, American approval polling is generally asked in two forms: job approval and general favorability. While job approval is cited more often - presidential job approval is very common - it only exists for candidates currently in office, so for previewing elections, favorability is the main measure used.

3.3.4 Pairwise

Pairwise polls consist of a series of questions, each pitting a single choice up against another single choice until all possible pairs of choices are exhausted. Polls of this sort are common for elections with a top-two runoff system, like in French or Brazilian presidential elections, but usually only candidates that are perceived to have a realistic shot of making the second round are included.

For example, in polls for the 2022 Brazilian presidential election released in March and April 2021 compiled on Wikipedia, the classic, single choice polls contain anywhere from four to ten choices. If these same polls had asked voter's preferences of every single possible pairing, they would have needed to ask respondents about their opinions on $\binom{n}{2} = \frac{n(n-1)}{2}$ pairings for n candidates. This translates to six pairings for four candidates and forty-five [3] pairings for ten candidates. Six pairings is a reasonable number of questions to ask a respondent; however, asking respondents to express preferences for forty-five different pairings is absurd proposition, especially when many of those possible scenarios appear to be essentially impossible. And consequently, within that specific selection of Brazilian polls, only eight different pairings were polled, all match-ups between the incumbent president, Jair Bolsonaro, and candidates with a reasonable possibility to be his challenger [12].

Because of these reasons, full polling of pairings between all candidates is extraordinarily rare, but that data is necessary for predicting and determining the Condorcet winner of an election, and for the use of most Condorcet electoral methods. Unless individual voters' preferences are cyclic (e.g. a voter preferring Candidate A over Candidate B, Candidate B over Candidate C, and Candidate C over Candidate A), pairwise preferences can be extrapolated from full ranked polling.

3.3.5 Summary of Polling Varieties

Since not all of these varieties of polling provide enough information to simulate every kind of election, each kind of input only triggers simulations of the electoral systems that it can do.

Single choice, approval, and pairwise inputs only trigger the simulations for the directly related electoral systems: FPTP, approval, and Copeland respectively. Ranked inputs, provided each ranked set has at least 2 of the choices ranked - if only one choice were ranked, it would be identical to a single choice poll - will trigger simulations for FPTP, RCV, Top Two, Approval, and Copeland - all the types of elections possible in this simulation. As ranked polling provides a high amount of information about voter preferences, it produces data that is either adequate in itself or is easy to extrapolate to estimate other measures.

4 Simulator

4.1 Structure of Algorithms

The algorithms are run via an overarching function `doAllSystems` that triggers the execution of a series of functions that execute the simulation for the kinds of election systems described in section 3.2. A visual representation of the layout of these functions is provided in figure 2.

To run a simulation using some input data, the function `doAllSystems` is run on a specific file that stores the information for the input data. This function reads the file, and then feeds that input data to *run* functions that handle the execution of simulations for each of the specific electoral systems: for example, `runFPTPElections` for first-past-the-post elections and `runApprovalElections` for approval elections. Inside each of these *run* functions, initially, each piece of input data on the file is checked to see if it is in the proper form to for this specific electoral system: for example, a file containing approval data will not work for a pairwise simulation whereas ranked data of depth four will work for all simulation types. If a piece of input data is deemed eligible to be simulated, the input data is fed into a specific *iteration* functions that run and

[3] Forty-four is a more reasonable number in this circumstance as two of the candidates polled, Luiz Inácio Lula da Silva and Fernando Haddad, are members of a single political party, the Workers' Party (PT), and since Brazilian political parties avoid running two candidates simultaneously in order to better their chances, it seems practically impossible that they would end up facing each other in the runoff.

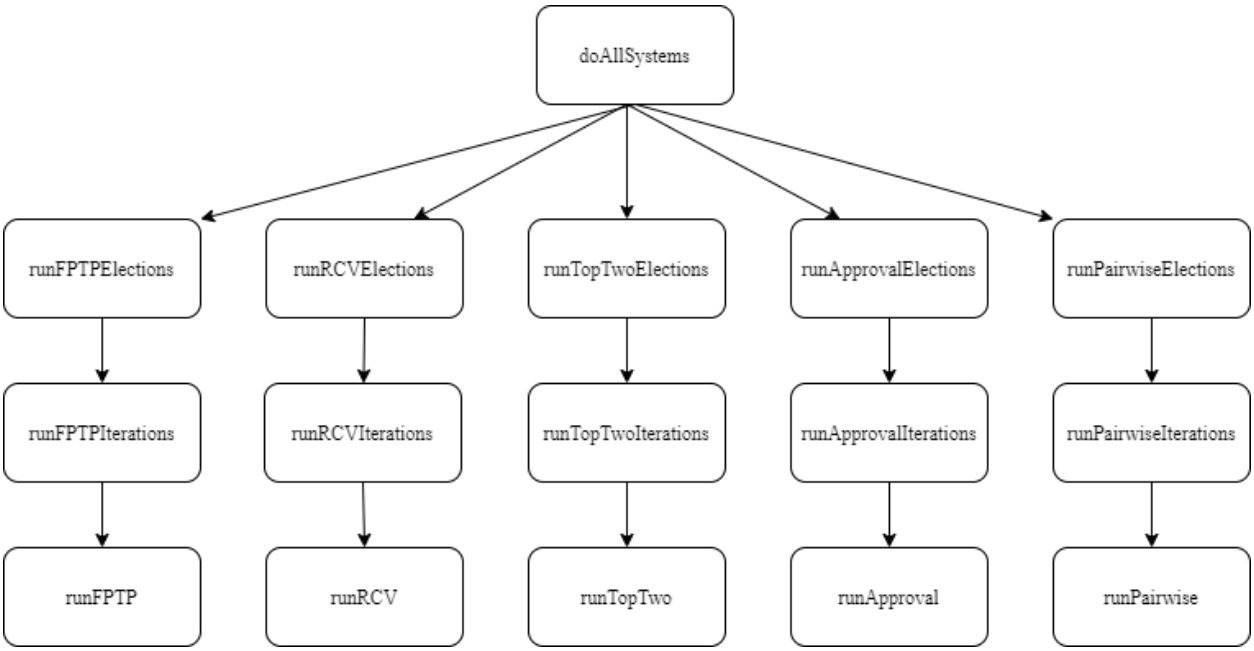


Figure 2: Overall Map of Functions of Simulator

compile the results of each iteration of the simulation. These take the form of `runTopTwoIterations` or `runRCVIterations`.

Inside these *iteration* functions, the input polling data is normalized to either a proportion between 0 and 1 (for approval and pairwise inputs) or as a proportion of the sum of the results of the poll (for single choice and ranked inputs). Then, an array within which the winner of each election iteration will be stored is initialized. Then, using a certain number of iterations will be executed for each simulation, the number of which is a developer-defined constant (currently set to 1000), based on the amount of time that each simulation takes to execute.

Within each iteration, initially, the polling data is randomly changed using the sample size of the poll (as described below) to simulate the random errors that can occur with election polling. Then, this randomized polling data (which is now the actual results for this iteration of this election) is fed into the final function for each algorithm (in the form of `runPairwise` or `runRCV`) which actually determine the winner of an election given the randomized results of this specific iteration. These functions output an index that represents the winner of this iteration. The electoral choice at that index is then given a win by incrementing its number of wins in the resultant array.

After all iterations have been run, this array is returned by the *iterations* function to the *run* function. A textual representation of this result is then added to a string containing textual representations of all of the current simulation results. A pie chart is also constructed based on this data and is saved to the server for future access. Finally, after compiling all the textual representations coming from each of the *run* functions, `doAllSystems` returns the combined textual representation which is then sent to and used by the frontend to display the results to the user.

4.2 Sample Size and Error

Key in the process of constructing an election simulation is including the possible errors of the polling data. There are two main types of error involved in polling: random error and sampling error.

Random error is simply a consequence of the limited number of respondents that a pollster talks to and

```
1
Melanchon,Hidalgo,Jadot,Macron,Bertrand,Dupont-Aignan,Le Pen
11,7,6,24,16,5,25
1730
1
```

Figure 3: Example of input file using data from polling for the 2022 French presidential election

the chance that a group of people will be slanted one way or another due to completely random chance, unrelated to any common factors like demographic qualities.

Sampling error is significantly more difficult to identify, and is essentially a consequence of a pollster choosing a sample of respondents that is not representative of the voting population. This discrepancy can often be caused by differing response rates between demographics or by the pollster simply having an inaccurate idea about the future makeup of the electorate. In this simulation, I am not correcting for sampling error as this is normally the pollsters' responsibility. For example, for its poll aggregation, the website FiveThirtyEight assumes that the pollster's turnout and demographic models are reasonable and does not attempt to modify the polls by changing their sample.

When a pollster runs a poll, they report the sample size of the poll, which ostensibly refers to the number of respondents to the poll, but is often a calculated number due to demographic weighting [4].

Using the reported sample size of a poll, one can calculate a margin of error for the poll, which is usually the radius of the range in which 95% - or two standard deviations - of the actual results should fall around the polled number. This means that each polled number has an error that can be calculated independently. Out of simplicity, for this simulation, I assumed that the polling errors were normally distributed, so for randomizing the input data, I used a normal distribution around the polled number for each choice. For this normal distribution, I need to find the standard deviation for each polled value. To calculate the standard deviation σ for a poll value x given a sample size s for the poll, the formula 2 is used.

$$\sigma = \sqrt{\frac{(1-x)x}{s}} \tag{2}$$

Using the mean value for the distribution, which is equivalent to the poll value x, and the calculated standard deviation σ, the simulator calculates a randomized value for the iteration. After the randomized value is found, the algorithm makes sure that the value is at least 0 and at most 1 [5], and recreates it is it is not within those bounds.

The simulator creates this randomized value for each polled value in each iteration of the simulation.

4.3 Input Data Syntax and Processing

The data files that are used by the simulation must be formatted in a highly specific format for them to work. The file needs to be a file with a txt extension. An example of this format is demonstrated in figure 3.

The first line of the file is the number of polls that are included in the simulation. This is largely for testing purposes as the user interface and the algorithms are not built to handle the user inputting multiple polls at once. If there were multiple polls included in a text file, then the next four lines will be repeated for each poll. Generally, this value will be 1, and there will only be a single poll included.

The second line of this file will be a comma separated list of the named of the candidates, parties, or choices in an election.

[4] Often, groups like young people or ethnic minorities are harder for pollsters to find as respondents, so many pollsters weight those responses more heavily than the responses of other demographic groups that respond to polls more readily. This over-weighting of certain voters reduces the effective sample size of the poll, and this modified number is generally what pollsters publicly report.

[5] These results would not be possible; no one can get negative votes or more than 100%.

The third line is a comma separated list of the integer-valued polling values. The exact meaning of each value is largely determined by the polling variety. For single choice and approval polls, the values directly correspond to the choices in the same position in the list of choices [6]. For ranked and pairwise polling, the correspondence is more complex. For ranked polling with n choices ranked, the each value corresponds to one of the permutations of the choices of length n. For pairwise polling, the values correspond to the length 2 combinations of the choices.

The fourth line in the file is the integer valued sample size discussed in section 4.2.

The fifth line in the file is a value that represents the varieties of polls that were discussed in section 3.3. An integer valued represents either single choice polling, when the value is 1, or ranked polling with n choices ranked for each entry, when the value is $n > 1$. An 'a' character represents approval polling and a 'p' character represents pairwise polling.

4.4 Simulation Algorithms In Depth

At the core of the simulation is the algorithms evaluating the winners of the electoral system. Each of these innermost algorithms - `runFPTP`, `runRCV`, `runTopTwo`, `runApproval`, and `runPairwise` - is radically different from one another, evaluating their electoral methods in different ways.

Each of these functions takes three parameters: `partyList`, `results`, and `depth`. `partyList` is simply a list of the names of the choices in the election. `results` is the list of adjusted (normalized to a fraction between 0 and 1 and then subsequently randomized for the specific iteration) polls in the same format as in the 3rd line of the input text files. As noted in section 4.3, the format of these poll lists is different depending on the variety of the polling data. `depth` is primarily meant to to represent the length of the ranked lists for the ranked polling data inputs, which I often refer to as the *depth* of the poll, as for all other inputs, it is set to 1. But since ranked is the most versatile input format and all the algorithm functions can use it, they all must have a parameter to capture the depth of the input polls.

4.4.1 runFPTP

Algorithm 1 Find the winner of an FPTP election

```
1: function runFPTP(partyList, results, depth)
2:     prefList = all permutations of the members of partyList of length depth ▷ List of lists that contain the list of ranked choices that correspond to each element of the results list
3:     partySums = empty array of length length(partyList) ▷ List that stores the proportion of the vote currently allocated to the choice matching the index
4:     for i = 0 to length(results) do
5:         index = index of partyList that equals prefList[i][0]
6:         partySums[index] = partySums[index] + results[i]
7:     end for
8:     return index of max(partySums)
9: end function
```

The function `runFPTP` simulates the first past the post electoral system using input data from either single choice or ranked polling. Since single choice is effectively a subset of ranked polling (just with a depth of 1), the algorithm works the same either way, looking at only the first ranking, regardless of the rankings' depth.

The algorithm functions as presented in algorithm 1. It begins by permuting the list of electoral choices into `prefList`, which stores each permutation of the choices as a list. This list is necessary to determine the outcome of a ranked poll, and sicne ranked polls work for each algorithm, this will appear in every algorithm.

[6] For example, in figure 3, the third name in the list of choices, Jadot, corresponds to the third integer in the list of polling values, 6.

Next, it creates an empty array to store the proportions of votes that each choice gets. Then, it fills that list by summing together the results that correspond to the `prefList` that have a specific choice ranked in first, then adding that summed number to the result storage array. Finally, it returns the index of the maximum of that result storage array.

4.4.2 runRCV

Algorithm 2 Find the winner of an RCV election

```
1: function runRCV(partyList, results, depth)
2:     prefList = all permutations of the members of partyList of length depth
3:     pointerList = array of length length(prefList) of 0s ▷ List that stores the index of the current top choice in each of the prefList elements
4:     partySums = array of length length(partyList) of 0s
5:     for i = 0 to length(results) do
6:         index = index of partyList that equals prefList[i][0]
7:         Add results[i] into partySums[index]
8:     end for
9:     while max(partySums) ≤ sum(partySums)/2 do
10:        index = index of partyList that has the smallest nonzero value in partySums
11:        for i = 0 to length(prefList) do
12:            if pointerList[i] is still in the bounds of prefList[i] and if the choice at index pointerList[i] in prefList[i] is the same choice stored at partyList[index] then
13:                increment pointerList[i]
14:                isTrying = true
15:                while isTrying do
16:                    if pointerList[i] is still in the bounds of prefList[i] then
17:                        newIndex = index of partyList that equals prefList[i][pointerList[i]]
18:                        if the choice for newIndex has not already been eliminated then
19:                            Add results[i] into the partySums for newIndex
20:                            isTrying = false
21:                        else
22:                            increment pointerList[i]
23:                        end if
24:                    else
25:                        isTrying = false
26:                    end if
27:                end while
28:                Subtract results[i] from the partySums for newIndex
29:            end if
30:        end for
31:    end while
32:    return index of max(partySums)
33: end function
```

The function `runRCV` simulates ranked choice voting using input data from exclusively ranked polling.

The algorithm functions as presented in algorithm 2. It starts off by creating a `prefList` as well as a `partySums`, just like `runFPTP`, but it also creates a `pointerList` that stores the index of the current top choice in each of the prefList elements. Since `runFPTP` never had to iterate through these lists, it never needed to keep track, but `runRCV` will need to do so. Then it will fill `partySums` with the values for the number of 1st place votes for each choice.

Then, the core of algorithm takes place in this while loop. The loop runs until the largest value in `partySums` makes up more than half the sum of all the values in `partySums`. As the algorithm eliminates choices and allocates their votes to extant choices, this possibility will get increasingly achievable because the number of nonzero elements in the list will keep falling. Then, it will find the index of the choice with the lowest support - this choice will be eliminated now. Then, it iterates through every preference list and if the choice that is being eliminated now is the current choice for a list (as indicated by `pointerList`), then it increments the value at `pointerList[i]` down the preference list until it finds an extant choice. If it finds an extant choice, it adds the value for this preference list - stored at `results[i]` - to the `partySums` value that corresponds to this new choice. If it iterates through the whole preference list without finding one that still exists, it just stops trying to add `results[i]` anywhere. Afterwards, it subtracts `results[i]` from the currently eliminated choice. Once it has iterated through every single preference list to reallocate all of the support for the choice, it should be eliminated, and the while loop repeats, eliminating another choice until one choice possesses over half of the remaining vote. And finally, it will return the index of that now-dominant choice.

4.4.3 runTopTwo

Algorithm 3 Find the winner of an Top Two election

```
1:  function runTopTwo(partyList, results, depth)
2:      prefList = all permutations of the members of partyList of length depth
3:      pointerList = array of length length(prefList) of 0s
4:      partySums = array of length length(partyList) of 0s
5:      for i = 0 to length(results) do
6:          index = index of partyList that equals prefList[i][0]
7:          partySums[index] = partySums[index] + results[i]
8:      end for
9:      topParties = array of choices with top two partySum values
10:     for i = 0 to length(prefList) do
11:         if the highest ranked choice in prefList[i] is not in topParties then
12:             increment pointerList[i]
13:             isTrying = true
14:             while isTrying and pointerList[i] is not out of bounds of prefList[i] do
15:                 if prefList[i][pointerList[i]] is in topParties then
16:                     Add results[i] to the partySum for that choice
17:
18:                 elseisTrying = false
19:                     increment pointerList[i]
20:                 end if
21:             end while
22:             subtract results[i] from the top ranked choice in prefList[i]
23:         end if
24:     end for
25:     return index of max(partySums)
26: end function
```

The function `runTopTwo` simulates a top two runoff election using input data from exclusively ranked polling.

The algorithm functions as presented in algorithm 3. Firstly, the algorithm creates those same three lists and fills the `partySums` list just like in the FPTP and RCV algorithms above (algorithms 1 and 2).

Then, it creates a list `topParties` of the two choices with the highest corresponding values in `partySums`. Then, it iterates through every preference list, checking to see which of the members of `topParties` ranks

higher in each one. If one of the members of `topParties` appears in the preference list, then the value in `results[i]`, which corresponds to `prefList[i]`, gets added to the index of `partySums` that corresponds to the higher ranking choice.

Then, finally, it returns the member of `topParties` with the higher value in `partySums`.

4.4.4 runApproval

Algorithm 4 Find the winner of an Approval election

```
1: function runApproval(partyList, results, depth)
2:     prefList = all permutations of the members of partyList of length depth
3:     partySums = empty array of length length(partyList)
4:     if input data is from approval polling then
5:         for i = 0 to length(results) do
6:             Add results[i] into partySums[i]
7:         end for
8:     else
9:         searchDepth = minimum of the length of the preference lists and the number of choices times a
   constant defined by the program
10:        for i = 0 to length(results) do
11:            for j = 0 to searchDepth do
12:                index = partySums index for choice in prefList[i][j]
13:                Add results[i] into partySums[index]
14:            end for
15:        end for
16:    end if
17:    return index of max(partySums)
18: end function
```

The function `runApproval` simulates an approval election using input data from approval polling or from ranked polling.

The algorithm functions as presented in algorithm 4. This algorithm creates a `prefList` and `partySums` like `runFPTP` initially. Then it checks the format that the input data is in. If it's in the approval format, the algorithm simply inserts the values from `results` directly into `partySums`.

If it's in the ranked format, then the algorithm initially defines a `searchDepth` value. This value is based on a constant defined in the program that indicates what proportion of the choices are approved by the voters in this circumstance. It is currently set to 0.5, so this means that the top half of choices will count as approved. But since a set of ranked poll results might rank fewer than half of the choices, in that case, every ranked choice would count as approved. This constant is the minimum of `depth` and the number of choices times that constant (currently 0.5).

Next, the algorithm loops through `results` and then loop up to `searchDepth` for each preference list. Inside this loop, it adds `results[i]` to the `partySums` index corresponding to each choice in the preference list up to `searchDepth`.

In either of these cases, the function just returns the maximum index in `partySums`.

4.4.5 runPairwise

The function `runPairwise` simulates a Copeland's method election using input data from pairwise polling or from ranked polling.

The algorithm functions as presented in algorithm 5. This algorithm, unlike the others, initializes a `pairList` and and a `superiorityList`, which are heavily analogous to the `prefList` and `partySums` respectively. `pairList` is every *combination* of length 2 of the members of `partyList`, and `superiorityList`

Algorithm 5 Find the winner of a Pairwise election

```
1: function runPairwise(partyList, results, depth)
2:     pairList = all length 2 combinations of the members of partyList
3:     superiorityList = empty array of length length(partyList)
4:     if input data is from approval polling then
5:         for i = 0 to length(pairList) do
6:             if results[i] > 0.5 then
7:                 index = index of partyList that equals pairList[i][0]
8:                 increment superiorityList[index]
9:             else if results[i] < 0.5 then
10:                index = index of partyList that equals pairList[i][1]
11:                increment superiorityList[index]
12:            end if
13:        end for
14:    else
15:        prefList = all permutations of the members of partyList of length depth
16:        for i = 0 to length(pairList) do
17:            partyAdvantage = 0
18:            party1 = pairList[i][0], party2 = pairList[i][1]
19:            for i = 0 to length(prefList) do
20:                if party1 and party2 are in prefList[i] then
21:                    if party1 is before party2 is prefList[i] then
22:                        Add results[i] to partyAdvantage
23:                    else
24:                        Subtract results[i] from partyAdvantage
25:                    end if
26:                else if party1 is in prefList[i] then
27:                    Add results[i] to partyAdvantage
28:                else if party2 is in prefList[i] then
29:                    Subtract results[i] from partyAdvantage
30:                end if
31:            end for
32:            if partyAdvantage < 0 then
33:                index = index of partyList that equals party1
34:                increment superiorityList[index]
35:            else if partyAdvantage > 0 then
36:                index = index of partyList that equals party2
37:                increment superiorityList[index]
38:            end if
39:        end for
40:    end if
41:    return index of max(superiorityList)
42: end function
```

is initialized the exact same way as `partySums`, except it will now just count up instead of being added to and subtracted from.

This algorithm has a similar structure to `runApproval` in that it first checks if the input data is pairwise.

If it is, then the algorithm will go the the algorithm will iterates along the `pairList`, checking the results that correspond to each entry. If the value is more than 0.5, than first choice in the pair has its corresponding `superiorityList` entry incremented. And conversely, the entry for the other choice in the pair is incremented instead if the value is less than 0.5. If the value is exactly 0.5, neither value is incremented, since this matchup was technically a tie.

If instead, the input data is ranked instead of pairwise, the algorithm creates a `prefList`. Then, it iterates through each element in `pairList`, and initializes a variable called `partyAdvantage` as 0. It also names the two members of the `pairList` element as `party1` and `party2`. For this `pairList` element, it iterates through `prefList`, and checks which of `party1` and `party2` appears first in the element. If `party1` appears first (or is the only one to appear), then `results[0]` is added to `partyAdvantage`; however, if it's `party2` instead, then it's subtracted from `partyAdvantage`. If neither appears, then `partyAdvantage` is not changed. Then, after all `prefList` lists have been iterated through, then if `partyAdvantage` is positive, then `superiorityList` is incremented at the index corresponding to `party1`, and if negative, the same for `party2`. After this is done for every element of `prefList`, then the index of the maximum value of `superiorityList` is returned.

If this maximum value is equal to $length(partyList) - 1$, then the winner is the Condorcet winner. For this function, if it must evaluate using ranked polls, it's worst-case runtime is on the order of $O(n^3 n!)$, which is very inefficient, especially compared to the runtime of the algorithm if approval polling is entered, which is on the order of $O(n^3)$.

5 User Interface

To allow users to interact with the web application, I created a user interface using out of javascript. First, this user interface allows the user to pick the variety of poll and method by which they want to upload the polling data to the server to be simulated. This will be done via a series of buttons which the user can select and unselect as they wish. Then once that is fully selected, that button array will be hidden, and the forms with which to input the data will appear, whether a file input or a text input. Then, once that data is fully input, it will be sent to the server, saved, and then summarized and sent back to the user for their approval. The forms will disappear and the summarized data input will be presented to the user with options to download the file, delete the file, or run the file through the simulation. If the simulation is selected, then after it is simulated and returned, a textual summary of the results combined with pie charts displaying the outcome will be displayed on the final screen, with an option to download the textual results.

5.1 Homepage and Header

As this website is not wholly devoted to the presentation of this web application, riggsmarkham.com serves a webpage with links to other parts of the site, specifically a link titled *election simulator* that takes the user to the subpage where the actual application is held. Also, on the election simulator application, a consistent header appears, providing links to go back to the homepage and also to reload the current page, which is the best way to reset it and wholly restart the simulation. The header can be seen in figure 4.

5.2 Selection Buttons

Initially, the user will see a series of buttons that can be pressed to choose from the various options available for simulating an election. The first button visible says *Simulate Election*, and when clicked, it reveals a button below it called *Polling Data*. When that button is clicked, the first real choice presents itself to the user. This is where the user will select variety of poll that they want to enter: single choice, ranked, approval, or pairwise. Once on of those is clicked, the rest of the buttons will gray out, disabled, and a choice

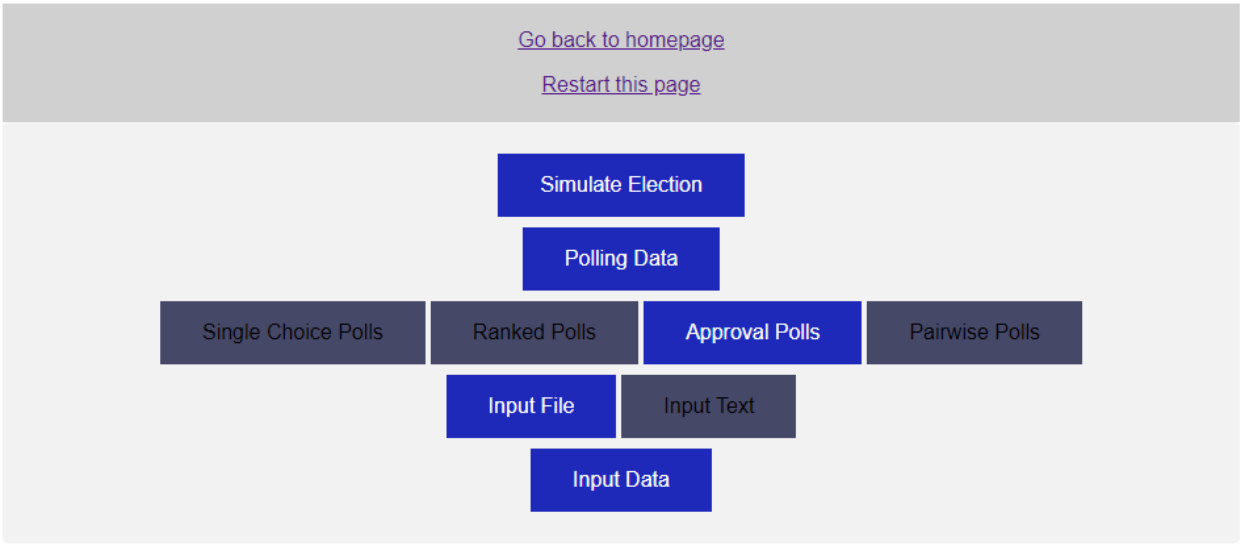


Figure 4: Layout of Selection Buttons & Header on Election Simulator User Interface

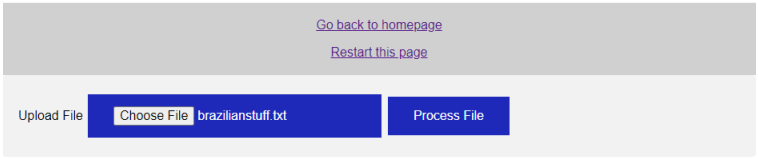


Figure 5: File Upload

of new buttons will appear below, asking for either file input or text format. Once one of those is clicked, the webpage asks for confirmation of the selection, and then moves onto the data input once this is clicked. An image of this section of the user interface is shown in figure 4.

5.3 Data Input

To input polls, there are two methods, file upload and text input.

The file input is very simple, as shown in figure 5. The upload will only accept text files and files under 1 MB as inputs. Once the *Submit* button is clicked, the file will be uploaded to the server and processed. File upload is far easier to use than the manual text input, so if one is planning on simulating the same poll repeatedly, it is recommended that they input the data manually, then download that file and upload it

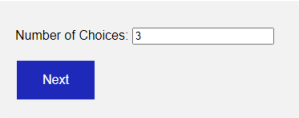


Figure 6: Text Input for Number of Choices

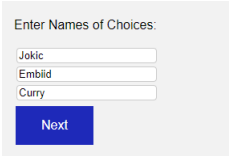


Figure 7: Text Input for Names of Choices

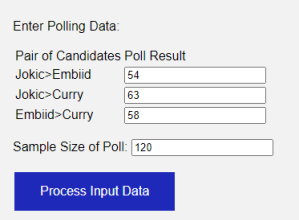


Figure 8: Text Input for Polling Data

whenever they would like to use that poll again.

The text input is much more complex, consisting of multiple screens of forms.

The first form asks for an integer number of choices in the election, as shown in figure 6. Once this is submitted, the web page presents a form displayed a number of forms equal to the number submitted in the previous form, as shown in figure 7. For the ranked polling format, this screen also asks the user how many choices will be ranked in each ranked list.

After entering these, the screen appears that allows the user to input polling data, as shown in figure 8. The specific layout of this screen differs for the polling variety selected. For the approval and single choice formats, it simply asks for the numbers that correspond to the individual choices. However, for ranked polling, the page asks for polling values for every single permutation of the ranked lists, and for pairwise polling, the page asks for every 2 length combination of the choices and for polling data for each of them. Just below this section, the application also asked for the sample size of the poll, as discussed in section 4.2.

Once these are input, the data is sent to the sever, where it is saved, processed, and then a summarized version is sent back to be displayed on the processed data page.

5.4 Processed Data Display

Once the data has been uploaded to the server, it is processed and returned to the client, displaying a summary of the input polling data according to the kind of input data that was selected, an example of which is shown in figure 9. This summary shows the number of candidates, the list of names of the candidates, the sample size of the poll, and the polling data, essentially in the same format as it was entered on the previous screen, with single choice and approval data sitting next to the candidates and the data for the others sitting next to the corresponding lists or pairs.

On this page, the buttons for running the simulation, downloading the file, and deleting the files off of the server also exist. The button for running the file, runs the file through the simulation on the server, which may take several seconds, especially with the ranked polling format. The download button will simply download the txt file on which the current poll data is stored, which is very useful if one wants to simulate that data again. The delete button will delete all of the files on the server that relate to the polling data in this iteration. After deleting, to add a new input, one must manually restart the page.

5.5 Presentation of Results

The results of an election will be displayed in two forms:

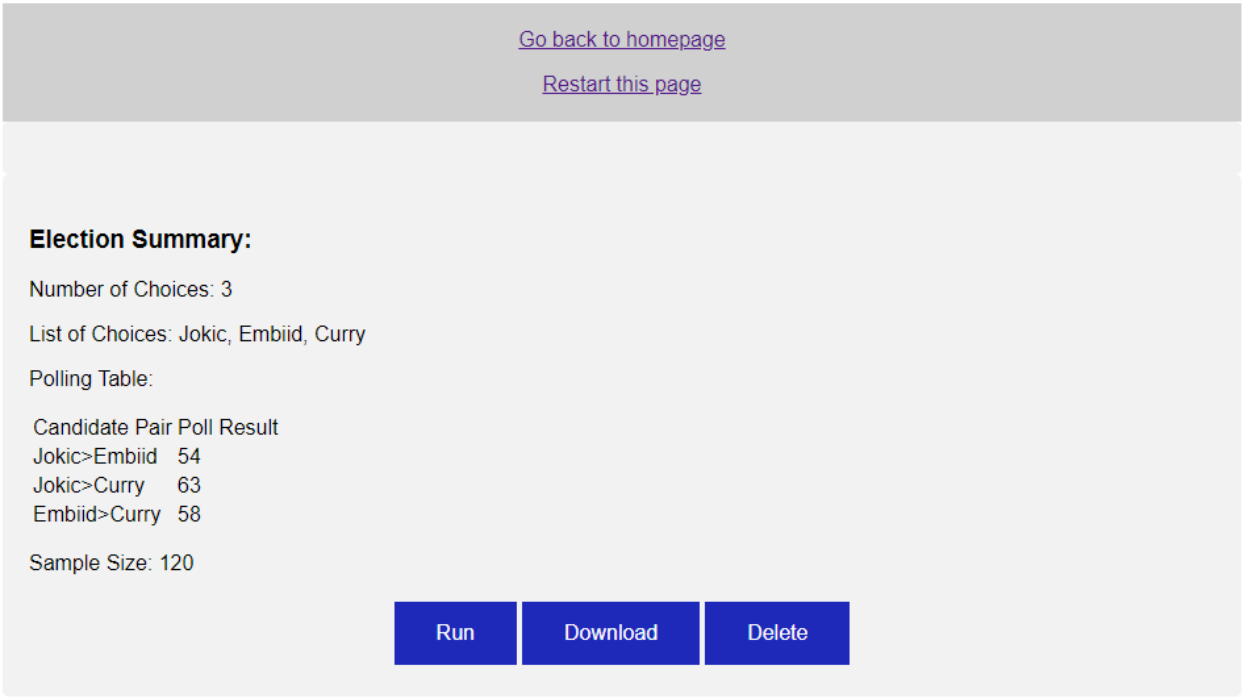


Figure 9: Processed Data Display

- Text Output
- Pie Plots

The text output is a string output directly from the simulation to summarize each of the electoral methods. The pie plots are made in the simulation and saved in order to be displayed on the webpage. Each of these results shows the proportion of total simulations that each choice won, with the pie charts grouping each choice that won less than 2% of simulations into an *Other* category. An example of the results of a simulation is shown in figure 10. The text output also shows the amount of time in seconds that each simulation took enumerated in seconds. If a poll input results in multiple simulation algorithms being run, the results of each algorithm will be displayed on this page. Additionally, if the user wants to download the results of this election in text form, they can click on the button.

6 Examples of a Potential User Path: Canada

Here, we will simulate through the path of a hypothetical user who is attempting to simulate the outcome of a Canadian election using a ranked poll of depth 2. In this poll, there are four parties, the Liberal Party, the Conservative Party, the New Democratic Party, and the Green Party as the choices.

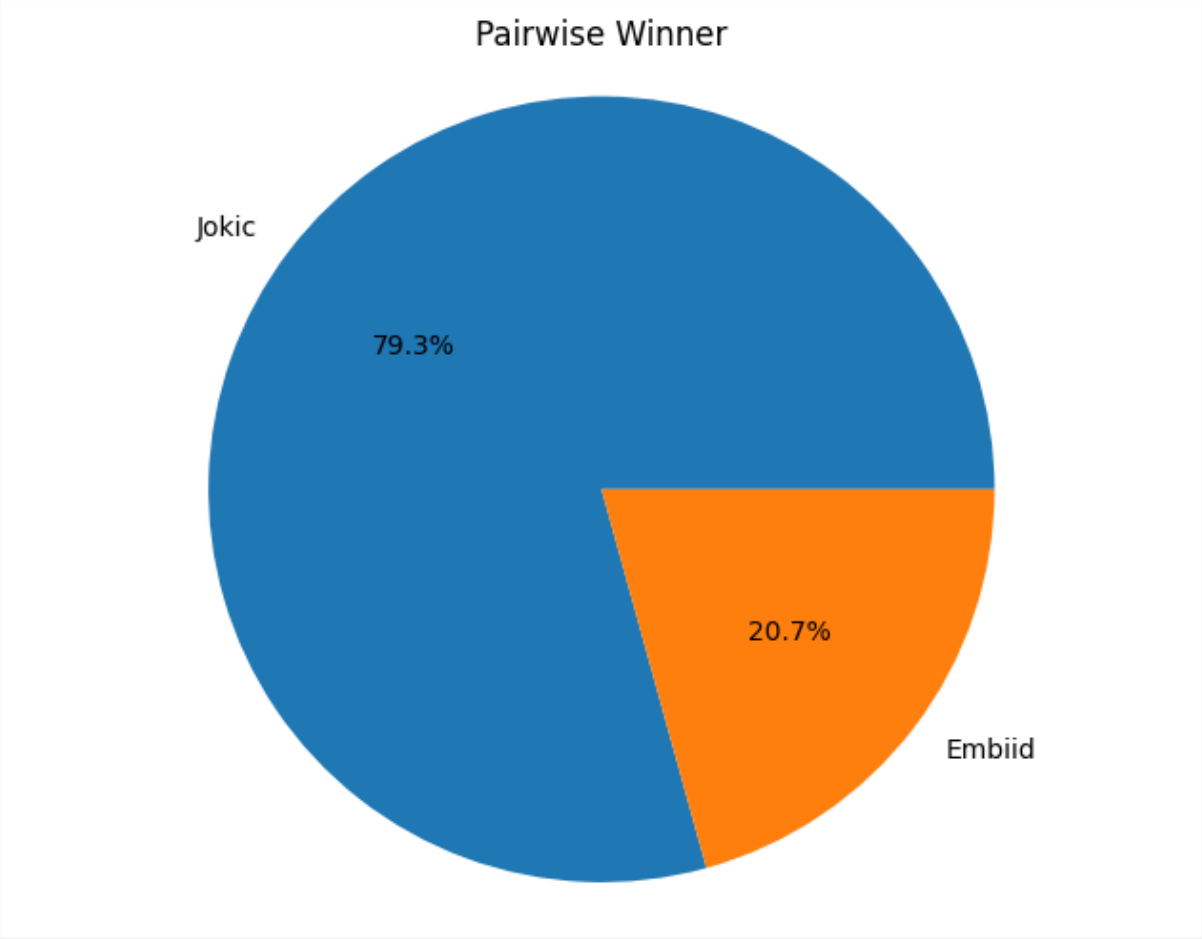
1. Navigate to entering ranked polling through the text input using the selection buttons (figure 11)
2. Enter the number of choices, 4, into the form (figure 12)
3. Enter the names of the 4 parties for this election (figure 13)
4. Enter the polling data for each permutation of parties (figure 14)

Simulation Results:

Simulation
Pairwise Elections

Jokic 79.3%
Embiid 20.7%
Curry 0.0%

Average time: 0.8027916099999999



Download Results

Figure 10: Simulation Results

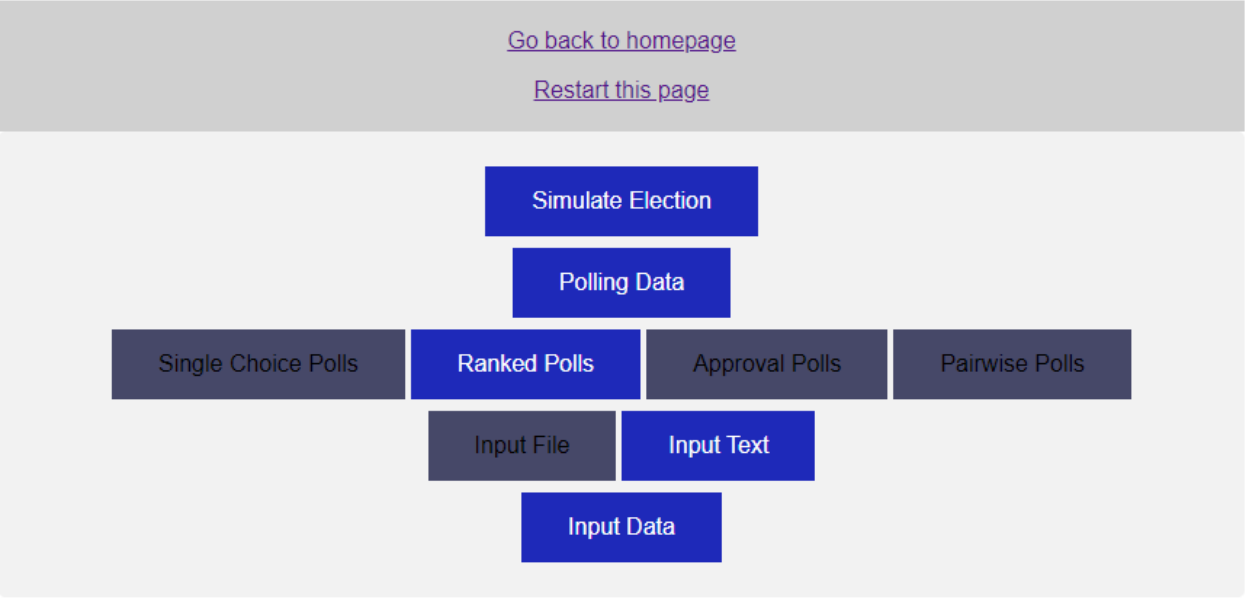


Figure 11: Canada: Input Type Selection

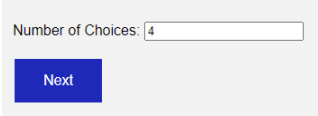


Figure 12: Canada: Number of Parties

5. Observe the processed data (figure 14)
6. Download the file for future use (figure 16)
7. Run the file and observe the output (figures 17, 18, 19, 20)

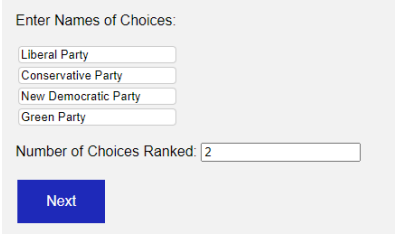


Figure 13: Canada: Names of Parties

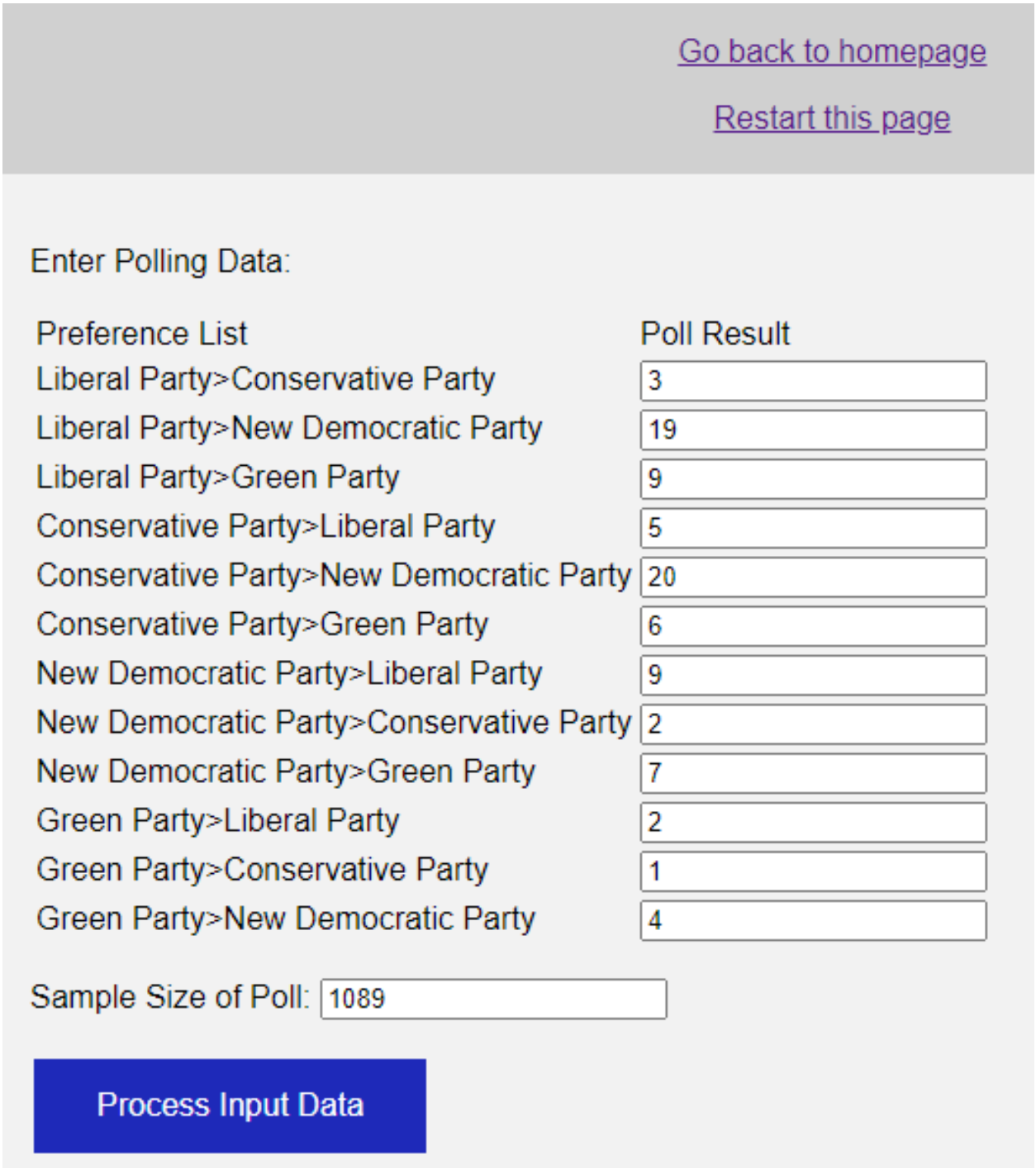


Figure 14: Canada: Polling Data Input

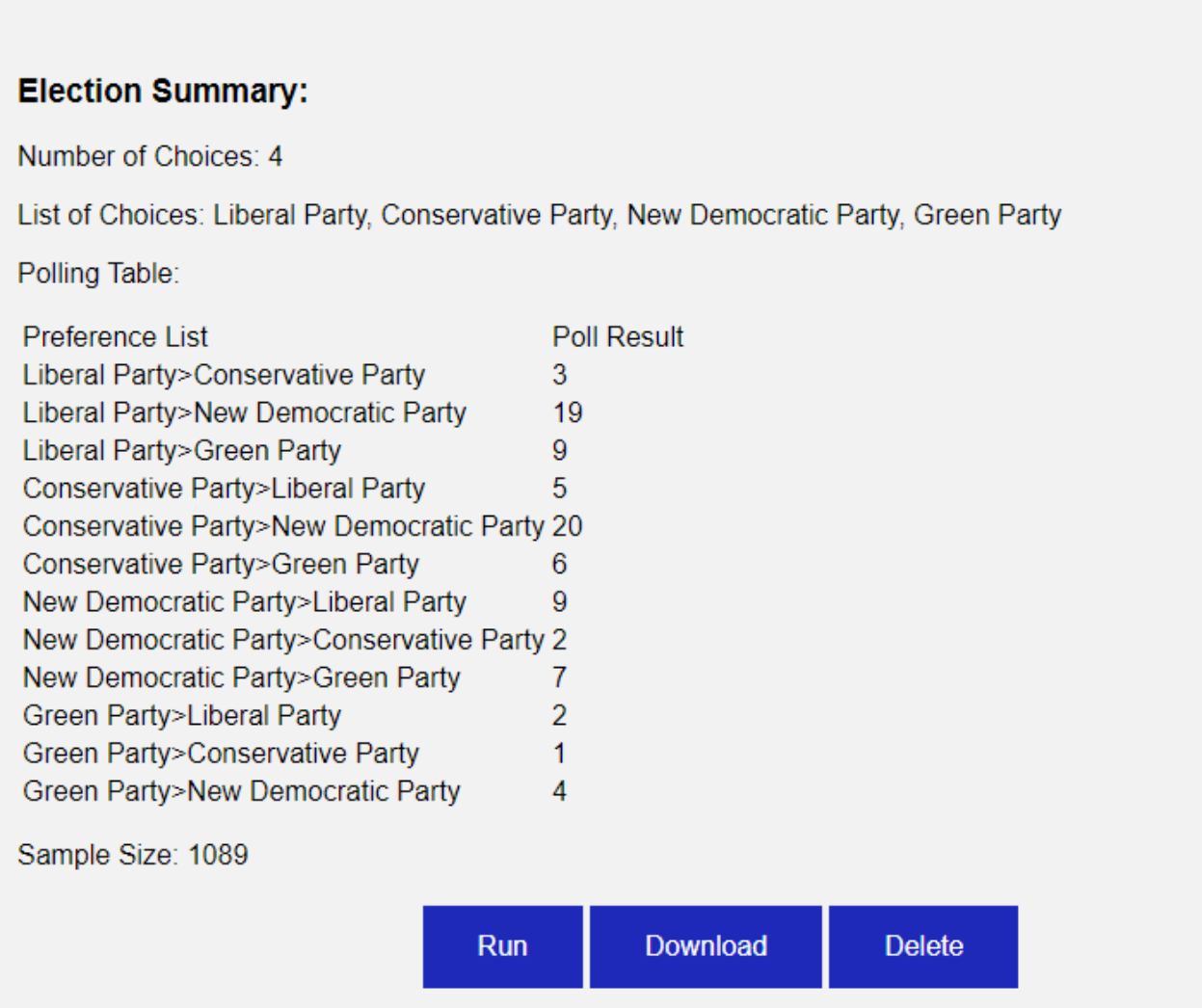


Figure 15: Canada: Processed Data

```
1
Liberal Party,Conservative Party,New Democratic Party,Green Party
3,19,9,5,20,6,9,2,7,2,1,4
1089
2
```

Figure 16: Example of input file using data from polling for a hypothetical Canadian election

Simulation Results:

Simulation
FPTP Elections

Liberal Party 47.2%
Conservative Party 52.8%
New Democratic Party 0.0%
Green Party 0.0%

Average time: 1.3216839390000006

RCV Elections

Liberal Party 100.0%
Conservative Party 0.0%
New Democratic Party 0.0%
Green Party 0.0%

Average time: 1.9651040860000002

Top Two Elections

Liberal Party 99.9%
Conservative Party 0.1%
New Democratic Party 0.0%
Green Party 0.0%

Average time: 1.6194265660000005

Approval Elections

Liberal Party 49.4%
Conservative Party 50.6%
New Democratic Party 0.0%
Green Party 0.0%

Average time: 1.2181678280000003

Pairwise Elections

Liberal Party 0.4%
Conservative Party 99.6%
New Democratic Party 0.0%
Green Party 0.0%

Average time: 4.2701057250000005

Figure 17: Canada: Text Results for Simulation

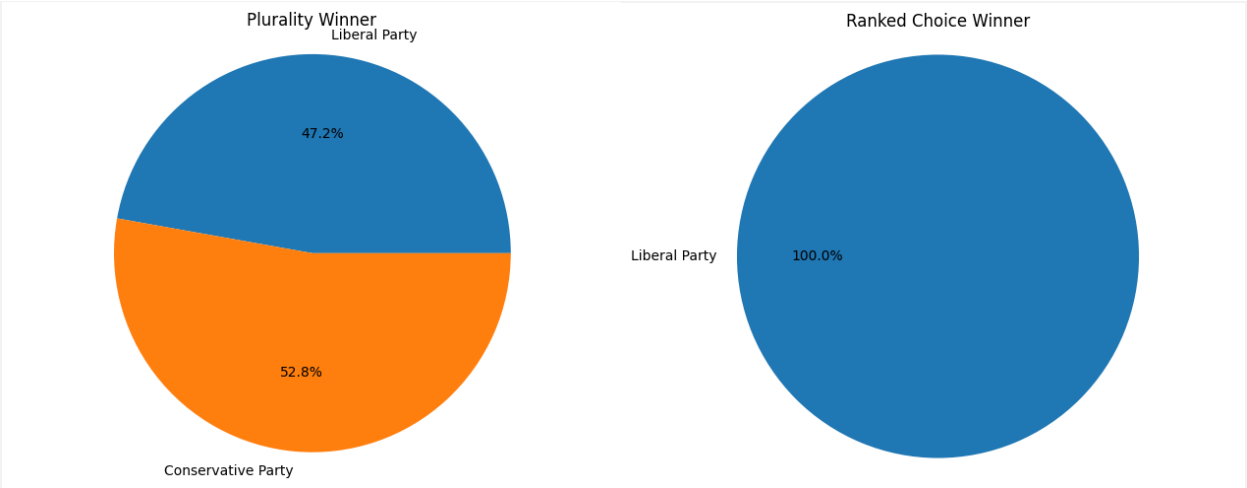


Figure 18: Canada: Pie Plots

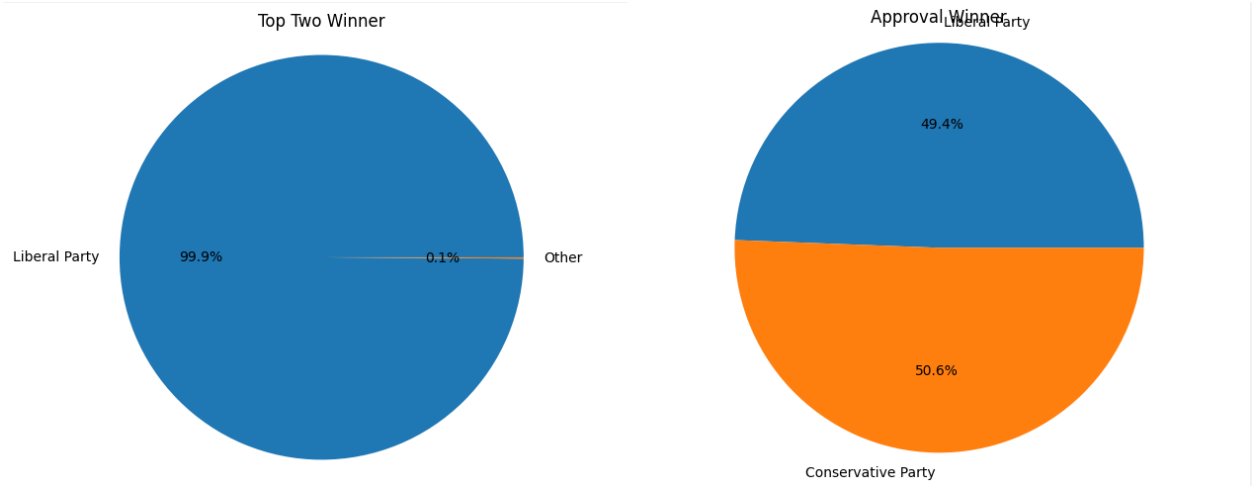


Figure 19: Canada: Pie Plots

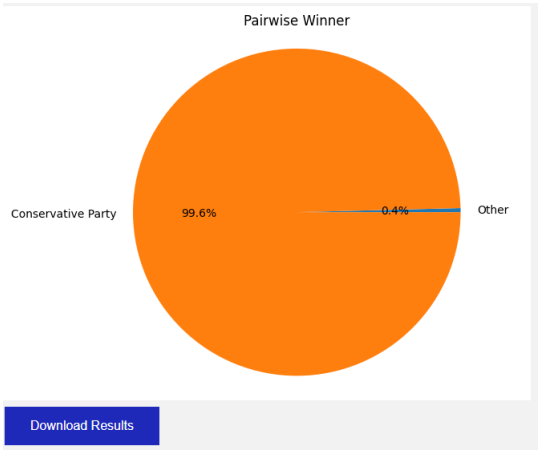


Figure 20: Canada: Pie Plots

7 Conclusion

7.1 Reflections

This election application serves the primary objective of being user-friendly to upload data and run electoral simulations. The combination of the well-formatted text input and the quick file input makes it easy and efficient to use. The button system on particular at the start of the web application is not only extremely easy to use for the client, but was also built in such a manner that it is extremely easy to expand from the back-end of the project. The forms, although sometimes time consuming, were well-organized and formatted, and the page showing a summary of the recently consumed data is especially informative. The file transfer system is as fast as could be expected when dealing with problems as computationally inefficient as elections.

On the other hand, the algorithms for evaluating the outcomes of the various electoral system are unoptimzed and slow, and this causes a great amount of confusion when having to wait for the algorithms to completely evaluate. Additionally, the randomized iteration system is rather simple and could be made more complex to improve the accuracy of the predictions of the application.

Overall, the application achieved its main goal of quickly and easily simulating elections for a user.

7.2 Future Work

The main area of future work is improving the simulations by adding additional electoral systems and making the existing ones more efficient. Additionally, creating more systems to estimate outcomes of various electoral systems even when the input data is not complete enough to normally evaluate them. General improvements of efficiency, security, and beauty could also be made.

Acknowledgements

I give thanks to Dr. Salan, Douglas Sims, and my parents for helping me immensely with this project.

References

[1] Yann Chevaleyre, Ulle Endriss, Jerome Land, and Nicolas Maudet. A short introduction to computational social choice. Technical report, Institute for Logic, Language and Computation, University of Amsterdam, 2006.

[2] Palash Dey. Computational complexity of fundamental problems in social choice theory. In *AAMAS '15: Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*, pages 1973–1974, 2015.

[3] Arnab Bhattacharyya and Palash Dey. Sample complexity for winner prediction in elections. Technical report, Department of Computer Science and Automation Indian Institute of Science - Bangalore, India, 2018.

[4] Andrei-Horia Mogos and Bianca Mogos. Comparison of voting methods using complexity functions and the weak theta asymptotic notation. In *2015 20th International Conference on Control Systems and Computer Science*, 2015.

[5] Michelle Blom, Peter J. Stuckey, Vanessa J. Teague, and Ron Tidhar. Efficient computation of exact irv margins. Technical report, The University of Melbourne, 2015.

[6] Lirong Xia. Computing the margin of victory for various voting rules. In *EC '12: Proceedings of the 13th ACM Conference on Electronic Commerce*, 2012.

[7] Donald Saari. Suppose you want to vote strategically. *Math Horizons*, 8:5–10, 2000.

[8] David Cary. Estimating the margin of victory for instant-runoff voting. In *EVT/WOTE'11: Proceedings of the 2011 conference on Electronic voting technology/workshop on trustworthy elections*, 2011.

[9] FairVote. How proportional representation elections work. `https://www.fairvote.org/how_proportional_representation_elections_work`, 2020. Accessed: 2020-11-02.

[10] Mostpha Diss, Eric Kamwa, and Abdelmonaim Tlidi. The chamberlin-courant rule and the k-scoring rules: Agreement and condorcet committee consistency. Technical report, Groupe d'Analyse et de Théorie Economique, 2018.

[11] Palash Dey, Nimrod Talmon, and Otniel van Handel. Proportional representation in vote streams. In *AAMAS '17: Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*, 2017.

[12] Wikipedia contributors. Opinion polling for the 2022 brazilian general election — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Opinion_polling_for_the_2022_Brazilian_general_election&oldid=1019697498`, 2021. [Online; accessed 9-May-2021].