CSCE 240
Intro. to Software Engineering

Voter Simulation: System Guide

Hadrian Buckner, Nicholas Grah, Brian Griffin, Peter Sanders, Tahir Warid

2016-12-01

Abstract

The purpose of this program is to simulate voter wait-times and determine how many voting booths are necessary in any given precinct in order to ensure voters do not wait inordinately long amounts of time to cast their vote.

1 Input Files

1.1 Configuration File

The configuration begins with a one-line header containing the following input data:

int Some arbitrary random number seed.

int$^+$ The length (in hours) of the election day.

int$^+$ The best-guess mean service time of a voter in minutes.

int$^+$ The minimum number of voters-per-precinct that will be supported by this simulation.

int$^+$ The maximum number of voters-per-precinct that will be supported by this simulation.

int$^+$ The maximum acceptable amount of time for a voter to wait.

int$^+$ The number of iterations of the simulation to run. As more iterations are run, the data becomes asymptotically more "complete" with respect to consideration of outliers, but the execution time grows linearly.

The following line(s) should contain one (1) more whitespace-delimited entries than the number of hours in the election day. These should be percent (%) values and should sum to one hundred (100). The first value represents the fraction of voters who were already in line when polls opened. The subsequent values represent the fraction of voters arriving in each hour of the election day.

1.2 Data File

This file is hard-coded into the program. The program must be executed from two directories below a directory containing `dataallsorted.txt`. In other words, `../../dataallsorted.txt` must contain the data file.

This file should contain whitespace-delimited integer values that represent voter service times in seconds. These values should be sorted in non-decreasing order.

1.3 Precinct File

The precinct file contains the definitions of arbitrarily many precincts.

Note: The file must contain nothing but complete precincts. Any superfluous text, any incomplete entries, or any invalid entries will cause the program to exit.

A Precinct is defined by the following information:

int Precinct ID. This is a *unique* numeric identifier for the precinct.

text Precinct Name. *Note: Must NOT contain whitespace.*

real Precinct Turnout. The percentage of registered voters in a given precinct who actually voted.

int$^+$ Actual number of voters.

int$^+$ Number of voters expected.

int$^+$ Expected rate of voters (voters per hour).

int$^+$ Number of voting booths.

real Percent of minority voters.

int$^+$ The last three items are used to set breakpoints. Detailed statistics are given as a histogram when the number of simulated voting booths is equal to the number given in these positions. Zero (0) is used to indicate the breakpoint is not used.

int$^+$

int$^+$

Only the number of expected voters and the precinct ID are used by the simulation. The other values are statistics that should be presented to the end user along with the simulation data.

1.4 Output Files

The last two arguments passed in the program call should be two file locations. The user executing this program must have write privileges at the specified location. The system must have on the order of megabytes of free space. Any files already existing at these locations will be overwritten.

The first file is used for the output of the program. The second file will also contain the output of the program, but if debugging options are turned on at compile-time, it will also contain log information. See Chapter 2 for detailed information on their contents.

2 Output Files

First, `MAIN` records the time and begins execution. Messages are printed to indicate the names of the output files.

Next, the configuration data is printed with the tag `CONFIG`. See §1.1 for more information.

`SIM` will be the next tag encountered. This is the `Simulation` class. It prints out the canonical string form of the `Precinct` being simulated. See §5.1 for more details on this string. As execution is passed off to this precinct, the next tag encountered is `OnePct`.

First, the canonical string form of the precinct is printed again. Next, the canonical string forms of the simulations for that precinct with one voting booth are printed. The number of simulations corresponds to the number of iterations specified in the configuration file. The canonical string form of a simulation may be found in Table 2.1. An example string follows, with superfluous whitespace omitted for the sake of space.

0 1 XXX00100 100 1 stations, mean/dev wait (mins) 0.43 0.95 toolong 0 0.00 0 0.00 0 0.00

This is repeated for as many iterations as were specified in the config file.

If the current number of voting booths was specified as a breakpoint in the Precinct File (See §1.3), a histogram is printed. One star on the histogram represents a computed value between one (1) and fifty (50) of voters. The histogram has a resolution of minutes and relates to the probability distribution function of voter wait time.

This, in turn, is repeated with incrementally more voting stations until there are no voters waiting too long.

Finally, this is repeated by `SIM` over the set of all precincts before returning execution to `MAIN`, printing the ending execution time.

Table 2.1: Canonical String Form of a Precinct Simulation

Simulation Number

Precinct ID

Precinct Name

Expected Voters

Number of Stations This is followed by the text “stations,”

Mean time to vote (mins) This is preceded by the text “mean/dev wait (mins)”

Standard Deviation of time to vote

Number of voters waiting too long This is preceded by the text “toolong”

Percent of voters waiting too long

Number of voters waiting much too long This is defined as ten (10) minutes longer than simply “too long”

Percent of voters waiting much too long

Number of voters waiting very much too long This is defined as twenty (20) minutes longer than simply “too long”

Percent of voters waiting very much too long

3 Main

This is the entry point in the Voting Simulation program. It is not a true class.

Main is responsible for gathering input and checking that arguments passed in are of the correct type. Main also opens the Utils log file before passing executi0on to the Simulation class.

Main does not contain any member functions or variables.

Main supplies input data to the configuration class and the MyRandom class, and passes data to the Simulation class that will be used to create the voting precincts.

Once Input files have been properly handles, an instance of the Simulation class is created, and RunSimulation is called for that instance of simulation.

4 Simulation

This class begins performing the simulation by creating the voting precincts from the input file, and calling the RunSimulationPct function for each precinct.

4.1 ReadPrecincts

Using a reference to the scanner reading from the input file, this function generates instances of OnePct and calls ReadData for that precinct, to populate it's member variables with values. This continues until the scanner stops providing input. These precints are stored in the pcts_ map.

4.2 RunSimulation

This function iterates through the pcts_ map, and, if the precinct has an appropriate number of voters for the simulation, calls `RunSimulationPct` for that precinct.

5 OnePct

The `OnePct` class is where the bulk of the work for this program happens.

Much of the data stored are statistics that are intended only to be presented to the user. The program does not use them anywhere as of this version.

This class depends upon the following public members of `Configuration`:

```
int election_day_length_hours_;
int election_day_length_seconds_;
int time_to_vote_mean_seconds_;
int wait_time_minutes_that_is_too_long_;
int number_of_iterations_;
vector<int> actual_service_times_;
double arrival_zero_;
vector<double> arrival_fractions_;
```

5.1 ToString

This defines the canonical string form of a Precinct. *Note: this is not the same as the canonical string form of a precinct simulation as described in Table 2.1.* The Canonical String Form has the following whitespace-delimited fields:

ID The integer-valued ID of the precinct. This must be unique within its batch.

Name This is a human-readable text identifier for the precinct. *Note: since fields are whitespace delimited, this cannot contain whitespace.*

Turnout This is a percentage (%) of voters who voted out of those who were registered for this precinct.

Voters This is the number of people registered to vote at this precinct.

Expected Voters

Voter Rate This is the mean number of voters expected per hour.

Booths This is the number of voting booths being used by the precinct.

% Minority This is the percentage of minority voters out of all voters.

HH This is the string literal "HH" and serves as the opening delimiter for the set of histogram breakpoints.

Break Point(s) These are the number(s) of voting booths for which a histogram should be printed. There may be up to three (3).

HH This is the string literal "HH" and serves as the corresponding closing delimiter.

5.2 ReadData

Access	`public`
Return Type	`void`
Arguments	`Scanner&`

This function accepts an open Scanner pointing at a *complete* standard serialization of an `OnePct` object. This is further detailed in §1.3. If the Scanner is not opened or is not pointing at a valid input string, the program will crash.

No member function may be called until this function has completed.

5.3 RunSimulationPct

Access	`public`
Return Type	`void`
Arguments	`const Configuration&`
	`MyRandom&`
	`ofstream&`

This function runs a series of simulations and analyzes their outcomes based on the data contained in the Configuration object.

Beginning with the minimum value that cannot be rejected trivially, it uses incrementally more voting booths in the simulation until none of the voters are waiting too long. This is done by creating a randomized set of voters with `CreateVoters` and passing off the processing of the queue to `RunSimulationPct2`.

Once that is complete, `DoStatistics` is executed and if this number of voting booths was specified as a breakpoint, a histogram is printed.

5.4 CreateVoters

Access	`private`
Return Type	`void`
Arguments	`const Configuration&`
	`MyRandom&`
	`ofstream&`

This uses random elements from a log-normal data set which should be made available to the program as specified in §1.2.

First a set of voters waiting at the door on poll open is created, and then an exponential random distribution is used to bring in a given rate of voters for each hour.

5.5 RunSimulationPct2

Access	`private`
Return Type	`void`
Arguments	`int`

This function accepts the number of voting booths which should be used by the precinct and attempts to process the queue. This program must be run after `CreateVoters`

5.6 DoStatistics

Access	`private`
Return Type	`int`
Arguments	`int`
	`const Configuration&`
	`int`
	`map<int, int>`
	`ofstream&`

This function performs analysis of the data from the simulations. It must be called *after* `RunSimulationPct2`. This function calls `ComputeMeanAndDev` and is the only valid context in which to do so.

6 OneVoter

The OneVoter class is used to represent a single voter to be used in the voting simulation.

6.1 Constructor

The Constructor for the OneVoter class takes in three integers as parameters. These integers assign values to the Voter's sequence_,time_arrival_seconds_, and time_vote_duration_seconds_ variables. The Time_Start_voting_seconds_ and which_station variables are set to default values 0 and -1 by the constructor.

6.2 General Functions

6.2.1 AssignStation

This function takes in station_number and start_time_seconds, and uses the data to calculate which station the voter will use, total time spent voting, and total time spent waiting to vote.

6.2.2 GetTimeDoneVoting

This Calculates time spent voting by adding time_start_voting_seconds_ and time_vote_duration_seconds_.

6.2.3 GetTimeInQ

Returns the time spent in the voting queue for a single voter by subtracting time_arrival_seconds_ from time_start_voting_seconds_.

6.2.4 GetTOD

Returns the time of day at which the voter voted.

6.2.5 ConvertTime

Returns the time with hours, minutes and seconds. Accepts the time in seconds from start of day that the voter voted. This function is called by `GetTOD`.

7 Configuration

The Configuration class is dedicated to reading in the input files detailed in Chapter 1. All varaibles are publically accessible, so maintainers should take care to always pass references to this object as `const`.

This class is a *Singleton*. A maintainer may wish to refactor it to this design pattern.

All other information has been covered elsewhere in Chapter 1

```
static const int kDefaultSeed = 19;
static const int kDummyConfigInt = -111;
static const double kDummyConfigDouble = -22.22;

// This is the seed to be used for the Random Number Generator
int seed_ = kDefaultSeed;

int election_day_length_hours_ = kDummyConfigInt;
int election_day_length_seconds_ = kDummyConfigInt;
int time_to_vote_mean_seconds_ = kDummyConfigInt;
int max_expected_to_simulate_ = kDummyConfigInt;
int min_expected_to_simulate_ = kDummyConfigInt;
int wait_time_minutes_that_is_too_long_ = kDummyConfigInt;
int number_of_iterations_ = kDummyConfigInt;
vector<int> actual_service_times_;

// This is the percentage of voters already in line when pools
    opened.
double arrival_zero_ = kDummyConfigDouble;

// This array will contain |election_day_length_hours_| elements
// which shall correspond to the percentage of voters arriving
// during each hour of the election day
vector<double> arrival_fractions_;
```

8 MyRandom

The MyRandom class is used to generate pseudo random numbrs from a seed provided. The functions contained in MyRandom are used by this program to generate voter data.

8.1 Constructor

The MyRandom class contains overloaded constructor functions. The first instance uses the integer 1 as the seed value, while the second takes an integer parameter value that is used for the seed. If a seed value is provided, MyRandom will use that value, but if a seed value is not provided, MyRandom will use a seed value of 1.

8.1.1 Seed Not Provided

```
MyRandom::MyRandom() {
  seed_ = 1;
  generator_.seed(seed_);
}
```

8.1.2 Seed Provided

```
MyRandom::MyRandom(unsigned seed) {
  seed_ = seed;
  generator_.seed(seed_);
}
```

8.2 General Functions

8.2.1 RandomExponentialInt

Parameters double lambda

Returns int r

Usage This function takes a double lambda as inpout, and returns an integer r. The lambda is the is the lambda value of the exponentially distributed real numbers. This function generates a double value based on the lambda constraint, rounds the double to the nearest integer, and returns this integer value. This function is used in the Simulation Program to calculate the interval of voter arrival times.

8.2.2 RandomNormal

Parameters double mean, double dev

Returns double r

Usage This function takes in doubles mean and dev for input, and generates a pseudo random number from a set with the provided mean and standard deviation.

8.2.3 RandomUniformInt

Parameters int lower, int upper

Returns int r

Usage This function takes in two integer s as parameters, describing the maximum and minimum values that can be generated. This function returns a pseudo-randomly generated value that falls between the upper and lower limits provided. In the Simulation Program, this function is used to calculate the time taken for a voter to finish voting. This function works well time taking to vote must be greater than zero, and less than the maximum service time specifed in the config file.

8.3 Private Members

unsigned int seed_ The value given to start pseudo-random number generation. This number can be provided when MyRandom is declared, or will be the default value 1.

std::mt19937 generator_ Pseudo-random number generator provided by the random library.

9 Execution Trace

The execution begins at an entry point referred to here as `Main` and procedes, most distally, to `OneVoter` through a topology described in Figure 9.1.

Execution begins at `Main`. First, all variables used by `Main` are declared. First, boilerplate program entry is taken care of: Arguments are validated, files are opened, system state is recorded, etc.

An instance of `Configuration` is initialized from the corresponding stream, which is then closed.

9.1 Configuration

`Configuration` is a *Singleton* collection of public variables that will be used throughout the program. It accepts an open stream for the configuration file, which should be of the format described in §1.1. Next, a log-normal data set is read in by the same call. This stream is opened in context by the `Configuration::ReadConfiguration` function. *NOTE: The program will crash if the file is not found.* The conditions for this file are given in §1.2.

From here, execution returns to `Main` and procedes to `Simulation`.

9.2 Simulation

Simulation does very little work and exists only to group `Precinct`s into batches. Precincts are read in from a file first, then execution is passed off to each precinct in turn to run its simulation.

9.3 OnePct

This is the "meat" of the program. Execution first enters `OnePct` at `RunSimulationPct`.

A simulation runs through a day of voting in order to determine the number of voting booths required to handle voters efficiently. The purpose of `OnePct::RunSimulationPct` is to regulate the number of voting booths being simulated. Given a mean voting time in seconds (MVT) and a number of voters (NV) on an election day (L) hours long, the initial number of voting booths (VB_0) is given by Equation 9.1 below.

$$VB_0 = \begin{cases} \lfloor \frac{MVT \times NV}{3600 \times L} \rfloor & MVT \times NV \geq 3600 \times L \\ 1 & \text{otherwise} \end{cases} \tag{9.1}$$

Beginning with this number and iterating upwards until no voters are waiting “too long,” the program runs the number of iterations indicated by §1.1. After all iterations have run, a histogram is printed if this number of voting machines is a breakpoint number for this precinct.

9.3.1 CreateVoters

Each single iteration consists of three parts. First, voters are created. This includes a set of voters waiting at the door when the polls open and exponential random voters at the hourly rates indicated by the configuration file.

9.3.2 RunSimulationPct2

Next, simulation is passed off to another function. This manages the processing of voters through the queue.

9.3.3 DoStatistics

Finally, some statistics are run to determine if any voters waited too long, among other things.

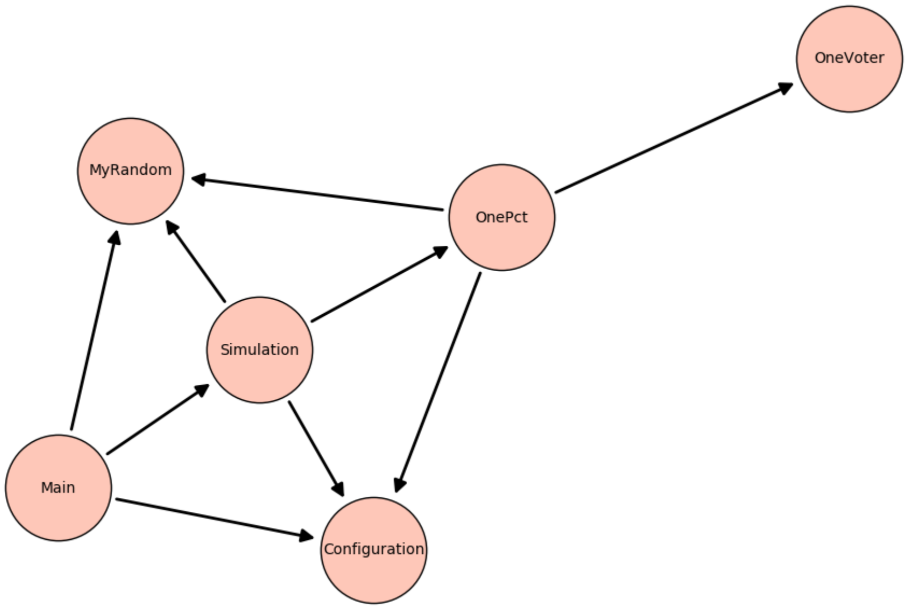

Figure 9.1: Execution Topology by Class