Group Project Report

Team 79

Pranay Raj Kapoor, Vipin Jain, and Jeff Grant

1. Additional Feature

Our additional feature returns the highest and lowest property market value for the zip code with the highest full vaccination rate per capita. We do this in four steps:

1. We use the the getFullVaccinatedPerCapita() method from feature 2, which uses the data from the population.txt and covid_data (.csv or .json) datasets to get the zip code with the highest the full vaccination rate per capita among all zip codes with population greater than 0 in the population dataset.
2. We then iterate over the list created by our PropertiesDataReader and for each PropertiesData object in the list. If the zip code field of the Properties Data object matches the zip code with the highest per capita vaccination rate from Step 1 above, we get the market value and then compare it to the highest and lowest market values for that zip encountered till that point (stored in the variables *max* and *min*). If the market value for the PropertiesData object is greater than the maximum value stored in max or less than the minimum value stored in min, we update the value to the appropriate variable (i.e. max or min)
3. Finally, we return a String that would be printed by the UI where the String has three lines. The first line in the String that we return is the zip with the highest per capita full vaccination rate. The second line in the String that we return is the max market value for that zip from the properties data and the third line in the String is the min market value for that zip from the properties data. If the properties data does not have any data for that zip, we print invalid for min market value and max market value in line 3 and 4 and if there is no zip we get from Step 1 then we just return a String with one line mentioning invalid or missing zip.

Because we used the modularity of our code to use a method from feature 2 in feature 6, our testing of feature 6 also included our testing of feature 2. We also needed to test the functionality of our Reader classes for each data set:

For feature 2, we used JUnit tests to check the functionality of the creation of the map from the two lists of data using the method on smaller lists of population and covid data we created that included regular and edge cases.

For feature 6, we used JUnit tests to check the functionality of iterating over the list of PropertiesData and finding the max and min value for the zip code with the highest vaccination rate calculated from feature 2.

To test the PopReader we compared our results of reading the given file in feature 1 to the stated correct result given in the project descriptions. We also used JUnit tests with sample data that contained edge cases to check the size of the resulting map and that the populations were mapped to the correct zip codes.

To test the JSONReaderCovidData and the CSVReaderCovidData classes we used JUnit tests with sample data that contained edge cases to check whether the CovidData objects contained the correct zip codes, number of partially and full vaccinated residents, and timestamps. We also compared the number of lines read by both the JSONReaderCovidData and CSVReaderCovidData and compared it to the covid_data file provided to see if there were any discrepancies.

To test the PropertiesReader, we used JUnit tests with sample data that contained edge cases to check the size of the resulting list and that the PropertiesData objects contained the correct zip codes, market values, and livable areas.

2. Use of Data Structures

In our program we used the following data structures: TreeMap, ArrayList, and HashMap.

1. TreeMap

   We used a TreeMap in our implementation of PopulationReader. We chose a TreeMap because we wanted to assign the number of vaccinated (partial or full based on user input) to each zip code and because we knew that we would want to return per capita vaccinations from feature 2 in sorted order by zip code. To do this, we stored the output from the PopulationReader in a TreeMap so that we could iterate over the zips in sorted order when implementing our algorithm for feature 2 and then add the output to the return String value of feature 2 in sorted order as well. Furthermore, a TreeMap was suitable as the Populations.txt data has one entry for each zip, so in the case of encountering duplicate zip codes in the Population.txt data whilst reading in PopulationReader, the PopulationReader throws an exception and the program terminates. For ensuring there were no duplicates in the Populations.txt data we felt a map would be appropriate given the data has one entry for each zip unlike the other two datasets. We use the the getFullVaccinatedPerCapita() method from feature 2, which uses the data from the population.txt and covid_data (.csv or .json) datasets to get the zip code with the highest the full vaccination rate per capita among all zip code's with population greater than 0 in the population dataset. However, most importantly the TreeMap was suitable to store the data as we could iterate through the map in sorted order and for each zip in ascending order get the per capita vaccination rate and add the result to the output string that feature 2 returns.

   We considered creating a List or Set of objects in a new PopulationData class to store the zip code, population, and vaccination rates, as an ArrayList or HashSet can use get() with O(1) complexity, but decided that this would use up more memory and would not lend itself to being easily sorted. So, due to memory reasons and efficiency in iterating over a TreeMap for feature 2, we felt a TreeMap was a good and efficient data structure to use.

2. ArrayList

   For the PropertiesReader, we used an ArrayList. Our sole reason for using an ArrayList is because the get() method for an ArrayList is 0(1). We chose an ArrayList over a LinkedList because of its efficiency in using the get() method to iterate over the list, since we do not anticipate removing any items from the list. Particularly, when we tried alternative data structures such as linked list, iterating through the list for features 3, 4, and 5 would take 10x more time since calling the get() method in a linked list is O(n) and thus in such a large dataset using an LinkedList would not have been efficient and met the time constraints. So given the actual performance we noted for the get() method in ArrayList, we chose an ArrayList. Particularly, using an ArrayLists over a LinkedList was adequate in meeting the time constraints for this assignment.

3. HashMap

   For the JSONCovidDataReader and CSVCovidReader class, we decided to store the information from the CSV and JSON files as a HashMap. We wanted to use a Map so that we could store the data in a way where the zip code would be associated with a list of CovidData objects that contain lists of CovidData objects with partial and full vaccination fields. We decided to use a HashMap instead of a TreeMap as the zip codes with which we are associating the data are already stored in a TreeMap created by PopulationReader, so adding an entry to a TreeMap O(log n), would be redundant and less efficient than adding to a HashMap O(1). Using a HashMap with an entry containing a list for each zip code is more efficient than using a list of all CovidData objects, as in feature 2, we can get the value for the given key (zip code) in the map and iterate through just the CovidData objects that are associated with that zip code to get the total number of people vaccinated instead of iterating over all CovidData objects created by the JSONCovidDataReader or CSVCovidReader. Thus given the efficiency in implementing feature 2 in this assignment, we used a HashMap for the JSONCovidDataReader and CSVCovidReader.

3. References used for regex for data management of the properties data filesare listed below:

https://stackoverflow.com/questions/18144431/regex-to-split-a-csv

https://stackoverflow.com/questions/1441556/parsing-csv-input-with-a-regex-in-java