Ritin Nair

CSVChecker README

Overview:

This project implements a pre screening check on a .csv file to determine if the values in each column match the designated type specified by a separate file. The prefix of each file name correlates to the type of values in each column. For example, take the following specification file:

```
A|3|int|String|int
B|3|char|String|String
C|5|int|int|int|int|int
D|1|char
```

The above file means that each file that starts with the prefix "A" should have 3 columns, with the first column having integers, the second column having strings, and the third column having integers as well. It is important to note that the specification file will be formatted properly, as if this were not the case it would not be possible to check if other files defined by the specification file have the proper format. The type of values in each column can either be a char, an integer, or a string. The file is iterated through to see if all the types match properly, and if not the row and column at which a mismatch occurred are outputted to a separate error file.

Implementation:

Standard Implementation:

The program first reads the file to be checked from user input following a prompt asking them to enter a file to be examined. The specification file details are then fetched based on the first character in the file name and stored in a string array. This includes the number of columns meant to be in each row and the type of the value in each column. The opencsv library is then used to parse through the file using a CSVReader object. A CSVParser object is also created to specify that the file will be delimited by the '|' character instead of the standard comma delimiter used for csv files. The reason this delimiter is used is because it is far less likely for the '|' character to appear in a string than it is for a comma to appear. Furthermore, the CSVReader is used to iterate through the file rather than a standard scanner and string split based on a delimiter because it provides the functionality of being able to use double quotes around any region where the delimiter should be ignored. For example, if an entry has a value of "hello|world" this is treated as one column instead of two being split up by the pipe delimiter.

The program goes through each row, and for each column in the row the type is checked against the specification file parameters previously stored in the string array to see if there are any mismatches. If there are, these are added to a result string to be later written to the error file all in one step. Doing one singular write is favorable to doing multiple as this avoids the IO overhead of making multiple write calls which could drastically affect runtime. The final result is then stored in a file called "error.txt".

For larger files, defined as a file with 100 or more rows, the file is partitioned into 4 separate subfiles where the file check is performed on each subfile individually on its own thread in parallel. The resulting error string value is then combined and written to the error file afterwards. Employing parallel processing allows for better scalability of this program by splitting the work among 4 separate threads running at the same time, and simply combining the result of each individual task to get the final result at the end. It is important to note that the threshold for determining what a large file is has been arbitrarily chosen, and a potential improvement could be to prompt the user at the start to input how they would define a large file themselves. Another way to determine the number of threads to use would be to manually test with files of different numbers of rows to find the point at which using multiple threads decreases the runtime by more than the amount of runtime added by the thread overhead.

Test Cases:

The project has a specification file included and multiple files which can be used with runTrial.java to test the program. The files all reside in the project directory, so to perform a test on the file called A_test for example, the user input for the file would be "./A_test". This applies to the other test case files as well, meaning that "./" should be prepended to every file name (unless the user wants to enter the absolute path of the file). The output of the file is in a separate file with the filename title and a .error suffix which will also appear in the project directory.

Runtime Analysis:

Let us say that there are n rows and m columns in the file to be checked. This program has O(n*m) runtime complexity. This is because each method iterates over the file once, and although the file may be iterated over multiple times (when counting the number of rows in the file to determine whether to use the multithreading approach for example) there are no instances where for each element every other element is checked in some way. Iterating through each column value in a row has O(m) complexity, and this is done n times for each row in the file, thus resulting in O(n*m) runtime. The iteration is done multiple times throughout the program, but this value is some constant k independent of the input size of the file. Therefore, k * O(n*m) once again simplifies to O(n*m) yielding in a final runtime complexity of O(n*m).

There are multiple reasons we decide to iterate row-wise through the files being checked versus parsing through them by column. Firstly, the CSVReader object has built-in functionality

to parse by row through the use of splitting the file into rows by newlines. This made the implementation far simpler than having to write my own parser that iterates column-wise instead. In addition, iterating column-wise through a file would be more inefficient when compared to iterating row-wise because for each column to get to the next row we would need to parse to the end of the row anyway to reach the newline where the row ends. Therefore, for each column entry we would need to also iterate over the row that the entry resides in (each row has around m entries so this is m operations to , and this would be done n times over all the rows of the file. This results in a runtime complexity of $O(n * m^2)$.

Challenges Faced:

One challenge I initially encountered was how to deal with a string that had a delimiter character inside it. At first, I was simply iterating through the file with a Scanner object and going line by line. For each row I would split the string by the '|' delimiter value, but this proved to be a problem in the case where a string had the delimiter inside it. To circumvent this issue I switched the approach from using Scanner to iterate the file to instead using a CSVReader as this object has built in functionality to account for this case. Another problem I faced was how to deal with empty values. I decided that even if a table has an empty value this does not necessarily mean that there is an error. For example, if a user's information were being stored in a table and the columns were meant to store their name, email, and phone number then not having one of these would not immediately invalidate the row or even be an incorrect value for the specific blank column. If this were meant to be an error, then it would be fairly simple to instead output this as an error in the error file instead. Furthermore, I was unsure whether or not a string could have invalid values or not. A potential example of an invalid string could be one containing a blacklisted word, but I did not account for this case as I felt that it was not that relevant towards the functionality of the program. I also decided that because any other type can be casted as a string that there would be no restrictions on whether the input was a string or not. For example, if the string to be checked was an integer value then this would still be valid because an integer can be casted as a string.