Aaron Kau and Kelvin Liu

README.pdf

This program, which parses all CSV files in a directory and its subdirectories, takes a list of labels to sort by, ordered by significance, and sorts the rows in the CSV and outputs it in corresponding sorted CSV files. The directory to sort and the output directory can also be specified. The program makes use of various system calls, functions, data structures to make the sorting process quick and efficient.

Usage

To run the code, first compile by running the *all* script on the supplied makefile.

1. *make*

Then, run the program by executing the following command.

2. sorter –c [column(s) to sort by] –d [input directory path] –o [output directory path]

where [input directory path] is the directory with CSVs to sort, [output directory path] is the desired location of the sorted CSV files, and [column(s) to sort by] is the comma separated column names to sort by. The -d and -o flags are optional. If they are not specified then -d is assumed to be the current directory and -o is assumed to output to the same directory as every CSV. Specifying the -d flag only sorts that one folder. Specifying the -o flag outputs all the sorter CSVs there. The CSV files are checked to have the specified 28 columns; if not, then they are skipped (note that the process is still created).

As an example, the *run* script (called by running *make run*) in the make file will sort the CSV files in folder, including movie_metadata.csv file (assuming it is in the same directory as sorter) by the column “director_name” and “gross”, and output the sorted CSVs in a folder called results in the main directory (with no subdirectories).

Assumptions

This program assumes that every file that is to be sorted has a name ending with .csv, and every subdirectory needs to have all its .csv files sorted as well. Since return values must be less than or equal to 255 and the way we count the number of children relies on the return values, we assume that there will be less than or equal 255 children created. Furthermore, there should not be any other filetypes besides basic files and folders, and all filenames are unique (an issue if all sorted files are output to the same directory and have the same name).

This program also assumes that the header file (Sorter.h) will contain the correct info about the CSV file's columns. The stringValues, intValues, and doubleValues arrays should contain the names of the columns that contain strings, integers, and doubles respectively (and stringValuesSize, intValuesSize, and doubleValuesSize should contain the length of these arrays).

The program will also initially allocate 5,000 bytes of space for each string and space for 10,000 rows in the CSV file (although it will automatically increase these global variables if they are exceeded).

The program will also assume the following about any null entries:

If the column stores strings, it is the empty string "".
If the column stores integers or doubles, it is the value zero (0 or 0.0).

Design

To loop through files in a directory we employ the use of the struct dirent, which is included in the <dirent.h> header file. This allows us to loop through items in a directory, including all the regular files and subdirectories. We ignore any other filetype, including the self and parent directory, and segregate the items into csv files and subdirectories. Each of these items gets a fork call, and the result of the fork is recorded in the main process when wait is called.

Each new process has to parse each CSV, which requires checking that there are the correct number of columns. If there are CSVs don't have exactly 28 columns, then the process ends.

To parse the CSV file, we created several structs that contain information about the file including the data stored, data types for each of the columns, column names, as well as the number of entries being stored. An enum called *type* was used for each of the possible data types that could be stored in the CSV (string, number, and decimal). The types for each of the columns in the CSV file were set beforehand using the F.A.Q. from the CSV file, although a CSV file with other column names and data types could be parsed by modifying the appropriate constants in the Sorter.h file (stringValues, intValues, doubleValues, and columns).

Each row in the CSV file was stored as an array of values. Since values could have differing data types depending on the column it resides in, value is saved as a union, which could be either an integer value, a double value, or a pointer to a string. The corresponding data type is retrieved from csv->columnTypes[columnNumber] to determine the type of that value.

Since multiple columns can be used to sort the entries, the query, for example [gross,movie_director], would be converted into an int array which holds the position of the headings, for example [8,2]. Whenever a comparison between two entries is done, the int array is used to compare the values at the indices specified by the array—the 8th position, and if the values are the same, then the 2nd position.

Mergesort

A mergesort algorithm was used to sort the CSV file by the specified column. The array of entries was split in half and the mergesort algorithm was recursively called on these subarrays. After these subarrays were sorted, they were merged together to form the fully sorted array using the algorithm to merge 2 sorted lists (Traverse through both lists with 2 pointers and keep adding the smallest value). Mergesort has a run-time of O(n*log(n)) with n inputs, where n is the number of rows for the CSV file.

Testing Procedures

Our program was tested by sorting various different combinations of csv files in various different number of subdirectories with varying column queries. We made sure to sort by each data type and we manually checked the output to ensure that in general, the resultant rows appeared to be in the correct order.

We also ran the program on directories already sorted to make sure that the CSV parser could take its own output as input and sort by another set of columns.

To check for CSV files of varying sizes, we also cut out varying number of rows to from the list to make sure that the CSV file could parse smaller CSV files, and we also appended the CSV file to itself (doubling the size of the CSV file) to see if the program could parse larger files.

Lastly, we changed several names of columns (and reflected such changes in our header file), and even changed the data types of many integer and double types to string and made sure the program was outputting correct CSV files based on these new column names and data types.

Challenges

Working with the fork system call was challenging because we could easily create a fork party(bomb) from an error in looping through directories and subdirectories. Also, understanding how to retrieve exit values from child processes was a bit difficult to understand as it is lumped together with another value in a single integer. Debugging with multiple different processes meant that gdb could not be used, and any print statements executed by the child would not appear synchronously.

We also faced a new challenge with regards to git. As we were working on multiple branches, there was often confusion when communicating what progress was made. There were also merge issues because of the different versions of code. In the future we will use a branch for each feature we implement.

Extra Credit

Extra credit 1 was implemented and included in the above description of the program. You can enter a comma separated list of column headings to sort by. This process is described in the Design section.