Script 1 Report (Group 15)

This guide finds itself divided for the two exercises. Each one's development will be described in detail.

A string with a length of 500000 was used. Although for most lines most of the memory allocated will be wasted, this allows us to read every line in the test files. This size was chosen because it's the length of the largest line, with some extra size to avoid running out.

The file is being analyzed linearly, line by line. The line is then either discarded or printed into the respective file.

Some auxiliary functions are used before the line is tested, checking if the line at least follows the basic format. The function find_errors is used to find empty fields in a line, when said line represent a user or a repository. A repository description or the message on commits can be empty and valid, so this test is not used on repositories. Despite analyzing a valid line twice, filtering the bad ones like this makes the overall program much more efficient.

The line is then put into a build function depending on the type of line.

As recommended by the professor, a lines' parsing is done with strsep, as opposed to strtok. This requires that the line pointer is copied twice. The original pointer is used for the line separation, thus it cannot be changed. A second pointer is used for the parsing, and a third pointer contains a copy of the line, unchanged, for printing into the file if it's copy passes the tests.

Even more auxiliary functions are used to make code simpler at this stage. The check function is used to verify if a parsed space is a positive whole number, check_Time checks if a date is valid and if it is not prior to April 7th 2005 and check_list verifies if a string is a valid Int array.

After the verification is completed and the line is valid, aforementioned copy of the line is inserted in the file using fprintf.

For the second part, the files resulting from the first part are opened, and the final files are created. After opening, two binary trees are created, one for the user IDs, the other one for the repository IDs.

Balanced binary trees and hash tables are very fast at finding values, however the group has more experience with binary trees, so it was the one chosen. The trees are created using the function build_Tree, the lines are read and parsed the same way as in the first part. The insertion and balancing is done by the file Tree.c, a modified version of the algorithm found in the link : https://www.geeksforgeeks.org/avl-tree-set-1-insertion/.

As the repository tree is created, the user ID is verified using the users tree, so when the program has to inevitably run the repository file a second time, it only has to check if there is at least one valid commit directed at a repository. This could have been done with a third tree, but the group decided that changing each node of the trees to include another integer for counting would be more efficient. With both trees created, the commits file is read, line by line, with the same strategy used in the first part, with the exception of the parsing being done all at once, and each segment being converted immediately and saved into variables. Since only the first two or three segments are being analyzed at this stage, this makes the code more legible. The trees are then searched for the values parsed. The added integer is incremented if searching for a repository. Finally, the repository file pointer is returned to the start of the file with fseek, the file is read again, the first segment is parsed and searched in the user tree, the second is searched and the calls to said repository is verified (the calls represent how many commits were aimed at which repository) with a modified copy of search_Tree, and the lines that pass all tests are printed in the respective file. After the files are closed, the program ends.