Gruffudd Trick [grt26] 190021683

CS23820 Assignment: “Analysing Sheep Behaviour from Tracker Tag Data”

Although not entirely complete, I managed to implement a majorit of what was asked in the brief to a good standard. The code is modular, using multiple classes for encapsulation and association to a single header file for structure. The main() function, if proved a collar data file, reads in the data, generates movement data, then prints to CSV. If no valid collar data is given, an advanced menu is printed, which allows tweaking of data boundaries in a configuration file. This is done to allow batch processing if required, or fine tweaking if needed.
The following sections outline the implementation of each class and their functions.

Task 1 - Reading Movement Data from file (COMPLETED)

My first task was to implement reading data from the provided collar plain text files and saving the data to a linked list. I implemented this in the class “reader.c”. The linked list is made up of structs called node_ptr’s. A filename is passed to a read_file() function. If the file doesn’t exits a NULL is returned. A while loop is used with the condition feof(file) == 0. This means the while loops keeps iterating until there is no more text in the file. The data is read in using fgets() and is stored in a char array called “line”. I used fgets over fscansf, because I needed the line for more than copying the data to variables, such as good data checking. An if statement then checks if the line is a valid line, by checking is the first element of the array is a digit character. If true the line is scanned using sscanf(), due to allowing the use of format specifiers, and the data format to pass the data to the appropriate variables in the node. If the linked list has no nodes, the current data is set as the head of the linked list, by calling make_node() which allocates memory to the node, then setting the head as the current node. Otherwise, if the position data is not equal to the previous data in the linked list, it is added to the tail of the linked list. The positions are compared for equality using the small function compare_prev(). The node is then added to the end of the linked list using insert_at_tail().

This part of the assignment was one of the most time consuming due to finding an efficient way of separating good and bad data and implementing the linked list for storing the data. That being said I didn’t particularly struggle with the implementation. One issue I did run into was my CMakeLists file not including the new class in the executable for some reason. However, it only took some googling to fix the issue.

Task 2 - Analysing the Data to Produce Movement Info (COMPLETED)

I implemented this stage of the assignment in the class “movement.c”. I used another linked list to the store the new movement data. To avoid data duplication and inefficient storage use, I used a pointer to the previous linked list to associate the data with their relevant coordinates. The majority of the implementation is in the function generate_movements(). It uses a while loop to iterate through positions linked list. Firstly the distance between start and end nodes is calculated in the function - “calculate_distance()”, which was bundled with the assignment. Then the duration is calculated using the calculate_duration() function. The function compares if the start and end time for size difference using if statements. I did this in order to avoid getting negative outputs for the time difference.
The speed is then calculated by dividing distance over time.
Similarly to the previous task, if the movements linked list is empty the node is set as the head, otherwise it’s insert at the end, using insert_movements_at_tail(). The node is allocated memory in the function make_movements_node().

Task 3 - Writing to CSV File (COMPLETED)

The third task is implemented in the class "csv.c". The filename and movements linked list are passed to a function called generate_csv(). This function creates a csv in the data folder. If a file of the same name already exists it overwrites the file. Once the file is created the function populate_csv() is called. This function populates the csv file with data in the format provided in the brief for the assignment by iterating through the linked list using a while loop. To write in the correct format I used the fprintf() function, as it allows the use of format specifiers to structure the output string.
If a config file (and not a NULL) is passed to the function it checks if each line of data is within the provided boundaries in the config data. This is done by calling the function is_allowed(). This config data was implemented in task 4.
Printing in the correct format was a moderately difficult task as it required an eye for detail, to ensure no mistakes in the format. However, I completed the task successfully.

Task 4 - Implementation of the config files (COMPLETED)

For the fourth task I returned to implement the configuration file. The majority of this was implemented in the class "config.c", with a menu interface in the main() function utilising the functions. The first function in the class, edit_config_data() allows the user to modify the data boundaries in the config, using a menu interface. This was implemented using a switch statement. Using a switch statement was useful as it avoids the need to use a large number of conditionals to read the user input. The second function, update_config_file() updates the config.txt file with the config data in the program. It simply uses a number of fprintf() calls to write the data in the correct format. The data is overwritten by removing the file if it exists, then creating the file, config.txt.
load_config_file() loads in the data from the config.txt. This was an awkward task as it required the use of strtok() to implement. I had some difficulty getting strtok() to work correctly. I opted to use strtok() as it allowed using delimiters to split the string and copy the data over. In addition, the structure of the config file is unlikely to change as it is only accessed by the program.
Lastly, there's initialise_config() which passes default values, specified in the brief, to the config data struct.

Task 5 - Implementing graphs (partially complete)

Using a modified version of the function provided in the brief, the function generate_lat_lon_speed_graph() generates a .gnuplot file containing plot data for use in the gnuplot program, to create a graph. Although I tried my best, I could not quite implement the graph correctly, to make it look like the provided example. The plot data is produced using a while loop to iterate through the data stored in the movements linked list. The data is printed in the format:
<start lon>, <start lat>, <change in lon>, <change in lat>, <speed>.

Conclusion

I believe I have completed a majority of the task to a good standard. If I were to repeat the assignment I would attempt to research more into Gnuplot in order to complete the last task. In addition, I would have liked to implement more thorough error checking. Although the program reads in the data files successfully, I believe it could have a more robust implementation, making use of more defensive programming.