**Carer Scheduling App – Code and Further Work**

**Sections of the code**

The code is spread across three different files.

The *cccmd.kv* file is a kivy style file. It contains information about the physical appearance of the app and includes functions that are called on specific events. The root widget is the WindowManager object, which includes the three different screens of the app: the main screen (FirstWindow), the carer view (CarerWindow), and the client view (ClientWindow). Refer to both the Kivy and KivyMD documentation on how to work with the widgets contained within each window.

The *functions.py* file is a collection of helper functions. These functions are what allow the algorithm to schedule and route carers.

The first section includes global statements. The api key is needed to use the OpenRouteService package and is linked to my account. It is encouraged to create your own ORS account and use your own api key to ensure this code continues to function in the future.

The remaining sections include general helper functions, functions that interact with real-life coordinates, functions that interact with the care time windows, functions that implement the clustering of points in multiple steps, functions that are involved in the creation of the carers’ routes, and a function that creates the final schedule. The functions themselves are annotated so that their purpose should be clear.

The *main.py* file is the main executable. It reads in the client and carer data and calls both the *cccmd.kv* file and the *functions.py* file. The global statements again use my api key, and they include information about Falmouth specifically. Add more lines here to adapt the code to work for other regions as well.

The internal function *setup\_dfs* creates the global clients and carers dataframes, which are used at several points in the code. There are two different types of data frames in this code. The *clients\_info* and *carers\_info* dataframes are pandas dataframes that contain the exact same information from the clients and carers information excel sheet. The only time the information in these dataframes is altered is when a carer’s or client’s availabilities are adjusted; these changes are only saved during the current session and are lost when Python is closed. The *clients\_df*, *clients\_df\_all*, and *carers\_df* dataframes take this information and reorganise it in a way the algorithm can use, which includes tracking changing carer location and number of hours worked. The *clients\_df\_all* dataframe includes all time windows during which a client should be visited, whereas the *clients\_df* includes dummy client IDs of one time window per client. The locations\_df dataframe is used for the visualisation of locations but is mainly a visualisation tool.

The majority of this code consists of interface functions. Every window is defined as its own class with its own functions. The FirstWindow class only contains three functions, which are clearly annotated. Most of the CarerWindow and ClientWindow functions are also annotated so that their purpose is clear.

The *update\_display* function of the CarerWindow class is the longest section of the class and therefore easy to break. An if-statement splits the function into three, based on which display type has been selected. There are commands specific to the type of display, such as plotting commands for the creation of a plot, but each section also includes hiding the other two output windows. These later lines of code are important; removing or messing with them results in different display types being shown at the same time, potentially interfering with each other. This function is called both when a new carer and when a new display type is selected.

The *update\_display* function of the ClientWindow works in a similar way. There is no map to display, so the if-statement only differentiates between two display types. However, both branches of the if-statement include updating the checkboxes that reflect the selected client’s availability. Although repetitive, these statements need to stay in their respective if-statements, otherwise the checkboxes do not update properly.

The CCCMDApp class is the main class of the app. The functions here interface with the file manager object from KivyMD. They allow the user to select the file that contains the carer and client information and then set up the global dataframes needed to run the code. Finally, the *build* function is necessary to make any kivy app function. It’s best not to change this part, as it may break the code.

**Further Features**

The following elements are still missing from the app. Here are suggestions on how to implement them:

* **Include other regions**: currently, only scheduling in Falmouth is supported by the app. The inclusion of other areas can be implemented by creating a separate csv file with each of the region’s postcodes, similar to the existing *tr.csv* file. In the app itself, the radio button selection on the first screen will then need to store the selection of the radio button selection (should be stored in FirstWindow.selected\_region), and select the relevant csv file based on this selection for the creation of carer routes.
* **Longer visits**: instead of True/False, the 5 time slot columns in the client information sheet should contain any number between 0 and 3, indicating the number of hours of care work needed in that time slot. An additional step may include allowing positive, non-integer values for these columns.
* **Only visit at indicated times**: Instead of assuming the client can be visited at any point during the day, they instead *must* be visited in their indicated time window. The code already functions like this for 1h long visits, but varying the length of a visit may break this, so this aspect needs to be included as well. To do this, the priority system needs to be reworked to account for this (*timewindow\_clients* in *functions.py*). The code already operates in a
* **Synchronised visits**: the fuzzy clustering already allows for clients to be assigned to two clusters. For a given time window when a synchronised visit should be scheduled, the client needs to be assigned to two carers. The route for one carer will dictate the route the other carer takes, to ensure they arrive at the client at the same time. In addition to this, the excel sheet needs to be adjusted to include a column/columns that reflect(s) which of a client’s visits are synchronised, if any.
* **PEG feeding**: both the client and carer information sheet should contain an addition column about whether PEG feeding is required for the client or whether the carer is qualified in this regard. For any time window, the clients that require PEG feeding should then be the first to be allocated to a qualified carer.
* **Walking Carers**: the travel time for walking instead of driving can easily be calculated using the OpenRouteService package. However, walkers and drivers will need to be treated separately during the clustering step, since the distance matrix cannot be calculated based on two different measures. I suggest for each clustering step in each time window to be split in two parts, with the less flexible walkers being scheduled first, with drivers then being scheduled to care for whichever clients remain.
* **Carers working multiple shifts**: A carer may clock in and out of work multiple times during the day. In theory, this is easily achieved by using dummy carers, with each dummy carer only having one shift a day. These dummy carers then need to be linked to the real carer with multiple shifts in some way; inspiration may be drawn from the handling of dummy clients in the code. In addition to this, the carer information spreadsheet will need to be adjusted to account for this.
* **Weekly schedules**: the creation of weekly schedules could be achieved by rerunning the daily program several times over. Depending on how the data is structured, carers may be assigned to specific workdays first. Overtime should also be tracked over the week instead of on a daily basis.
* **Fix the map display**: currently, the map of a carer’s route uses dots to show the route a carer travels. The code includes unsuccessful attempts to connect these dots for a cleaner interface, but further work can be done here.