Keyboard Analysis

Keshav Saravanan, EE23B035

October 2024

In this assignment, I implement a keyboard heat map similar to as seen in this demo.

1 Running the Project

To run the project, place the python files in the same directory. There are two ways to run the project - using the python script, or in the Jupyter notebook. The python script saves the final heat map as an image file. The Jupyter notebook contains an animation of the heat map being updated as characters are read from the input text.

To run the python script, run `python ee23b035_main.py`. There are optional command line flags for this: `-f` to specify the file to read the input text from, `-o` to specify the output image file, and `-i` to ignore spaces in the input text (since spaces occur very frequently, they tend to reduce contrast between other letters). The heatmap is saved as a separate image file

To run the Jupyter notebook (which contains the typing animation), specify whether or not to ignore spaces. The input text is also specified as a multi-line string. The heat map animation is displayed in the Jupyter notebook.

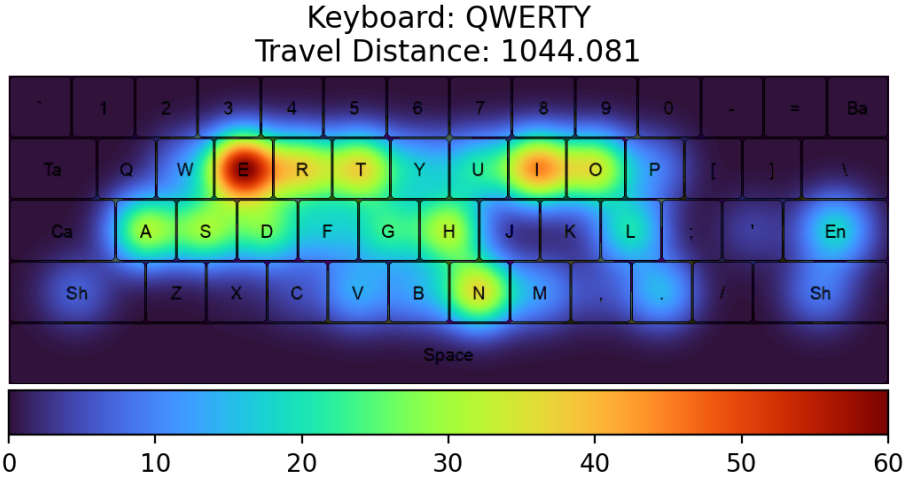

Figure 1: Heat map for sample input text on the QWERTY layout

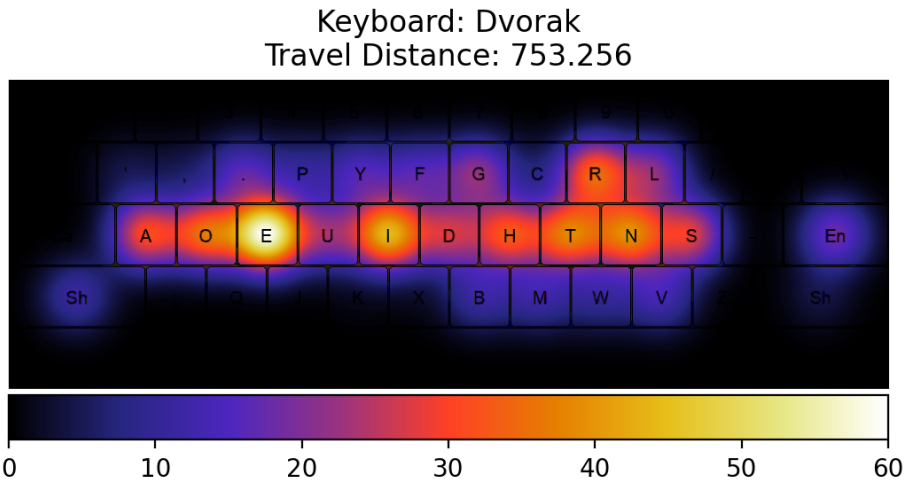

Figure 2: Heat map for sample input text on the Dvorak layout

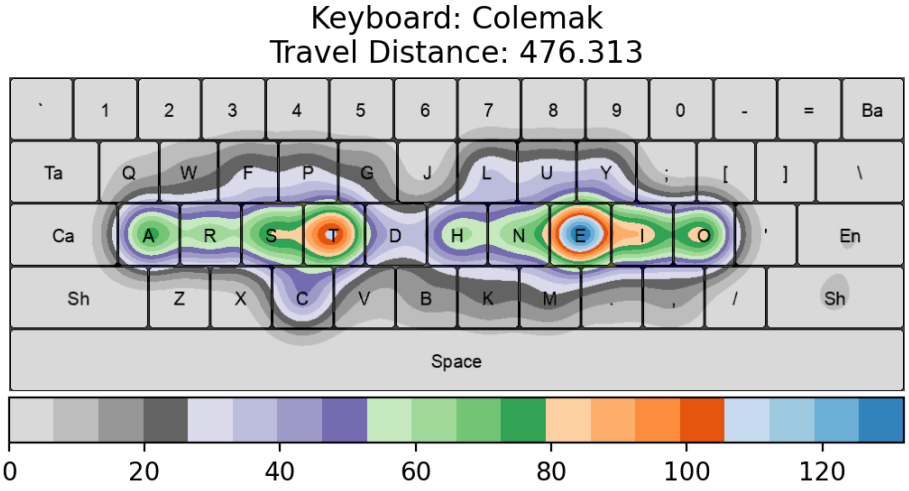

Figure 3: Heat map for sample input text on the Colemak layout

2 Configuration

The heat map, and keyboard layouts can be configured by the user. To change a keyboard layout, go into `ee23b035_keyboard_layouts.py` and configure the heat map, edit `ee23b035_config.py`.

The keyboard layout, and name can be set with the `KB_LAYOUT` and `KB_NAME` parameters. The output image's dimensions and font can be configured with the relevant constants.

The home row positions are defined as an array of 8 position coordinates - each coordinate for a finger.

Finally, the colour map and heatmap "spread" (defined below) can be configured with `CMAP`, `G_WIDTH` and `K_PRESS_HMAP_W`.

3 Approach

3.1 Calculating Travel Distance and Key Press Frequency

Calculating the travel distance is straightforward - First, given a character to type, I obtain a sequence of keys to press, and the calculate the shortest possible distance to each key from each of the home row positions. Summing it all up, this gives me the total travel distance.

I use the old keyboard layout format to store information about the keyboard. This means, that when typing a character, my script first checks if the given character can be typed with a single keystroke, or would require the use of the shift key. If the character requires a shift key, the shift on the opposite side of the keyboard is chosen.

By not mapping the keys to fixed home row positions in the layout configuration, it is possible to work with dynamic keyboard layouts. In theory, this means that the home row positions need not even be centered on keys in the keyboard - they could be placed according to the user's requirements. Also, changing a keyboard layout does not necessarily mean the home row positions would have to be changed, since the finger positions would likely remain the same.

While calculating the travel distance for each character, the script also updates key press frequencies. At the end of execution, this information is printed out as a mapping of keys to frequencies.

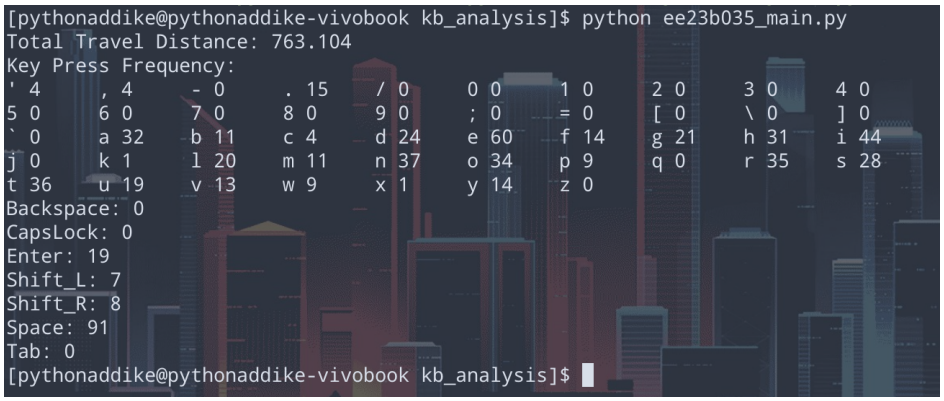

Figure 4: Key frequency for sample input text

3.2 Drawing the Keyboard

The keyboard layout must satisfy some basic rules - all required keys (alphabets, numbers, special characters, and a few special keys) must be present. Any regular key must also have a "shift character" associated with it - the character that is typed when the shift key is pressed. In addition to this, the first row must have row number 0, and empty rows are not permitted.

I draw the keyboard using the coordinates provided in `KB_LAYOUT`. Each coordinate is treated as an indication of the row number and left end of the key - so the y-coordinate must be an integer.

This results in slightly less information than is required - the width of the rightmost key will not be known. To resolve this, I assume the left-most and right-most keys have the same width, and accordingly position them so that the row width is constant across rows, resulting in a neat keyboard.

3.3 Creating the Heat Map

Every time a key is pressed, I add a Gaussian distribution to the heat map array, centered at the location of the key. This creates a smooth distribution of color across the keyboard. The spread of the Gaussian is controlled by `G_WIDTH` and `K_PRESS_HMAP_W`. Increasing `G_WIDTH` results in the the spread of the Gaussian decreasing, since the the Gaussian is sampled across the range `-G_WIDTH` to `+G_WIDTH`. Increasing `K_PRESS_HMAP_W` increases the area over the keyboard which is affected by a key press.

This approach has one drawback though - a key's colour can be updated by nearby keys, especially if there is a nearby key with a large key press frequency. This effect is exaggerated with strings of very short length (for example, with `"ffssdooo"` in the QWERTY layout). However, with longer strings of text (50+ characters or so), this effect disappears.

The color map can be changed by changing the `CMAP` variable - the default is `"rainbow"`.

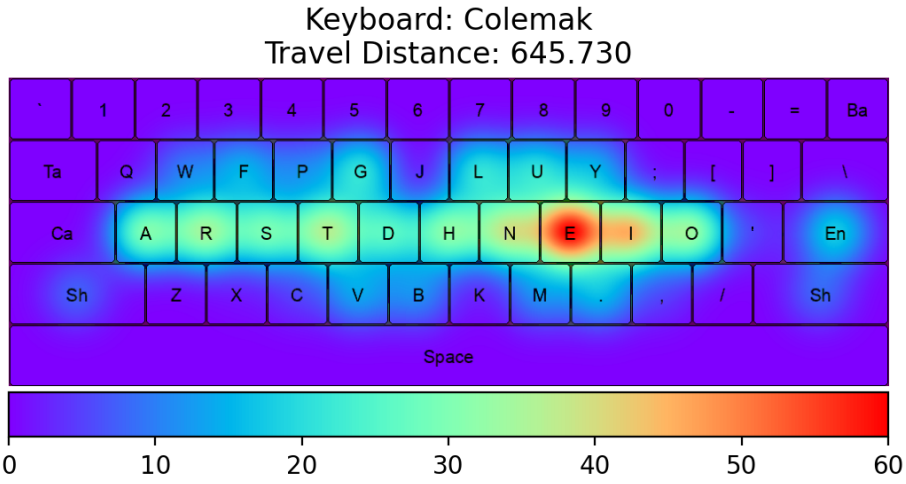

Figure 5: Heatmap with `"rainbow"` color map

4 Comparing Keyboards

Colemak seems to be the most efficient layout out of the 3 I tested - the heatmap is fairly concentrated along the home row (in Dvorak, keys like R and U could be swapped to obtained a more efficient layout). This can also be seen by the fact the travel distance for long texts in Colemak generally turns out to be lower than the same on Dvorak and QWERTY.