The Braille Typing Companion

The braille typing companion is a mini prototype of a project aimed at solving real-world QWERTY keyboard to braille-based typing issues. I'm glad to have gotten the opportunity to partake in Thinkerlabs' creative coding task. This project was overall a great experience for me to brainstorm different ideas and research more into braille literacy and the importance of integrating technology to revolutionize and improve accessibility.

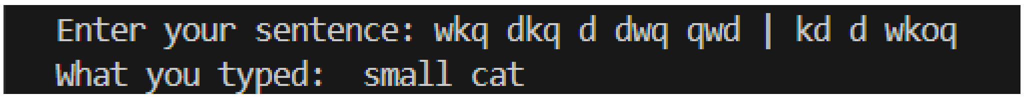
The document below contains a summary of all the functions, optimization methods and step by step improvements I have made on each version of the code to improve it and finally produce the best possible output. Unfortunately, since the task synced it's alignment with my final exam schedule and was required to be submitted at the earliest, I tried my best to incorporate most features I had in mind for the prototype- but still had to miss out on a few enhancements which I will further explain in the document.

Making of the prototype:-

Version 1:

I set up a limited word dictionary and set up a module called 'qwerty_to_braille' for converting the typed characters to their appropriate alphabet. A **frozenset** was used in handling character conversion because the letters when pressed simultaneously can be in any order.

The core of this code is choosing which main algorithm should be use for the autocorrect system. My options were between the Levenshtein distance algorithm and the Trie tree approach. I eventually decided to go with the Levenshtein distance algorithm for suggesting the word based on the least distance to the typed in word because of its simple yet strongly effective implementation. All the words of the same distance were stored in a library and one out of them was randomly chosen.



```
Enter your sentence: wkq dkq d dwq qwd | kd d wkoq
What you typed:  small cat
```

Though this worked fine for a basic autocorrect system- it missed several important functions to be more accessible.

Version 2:

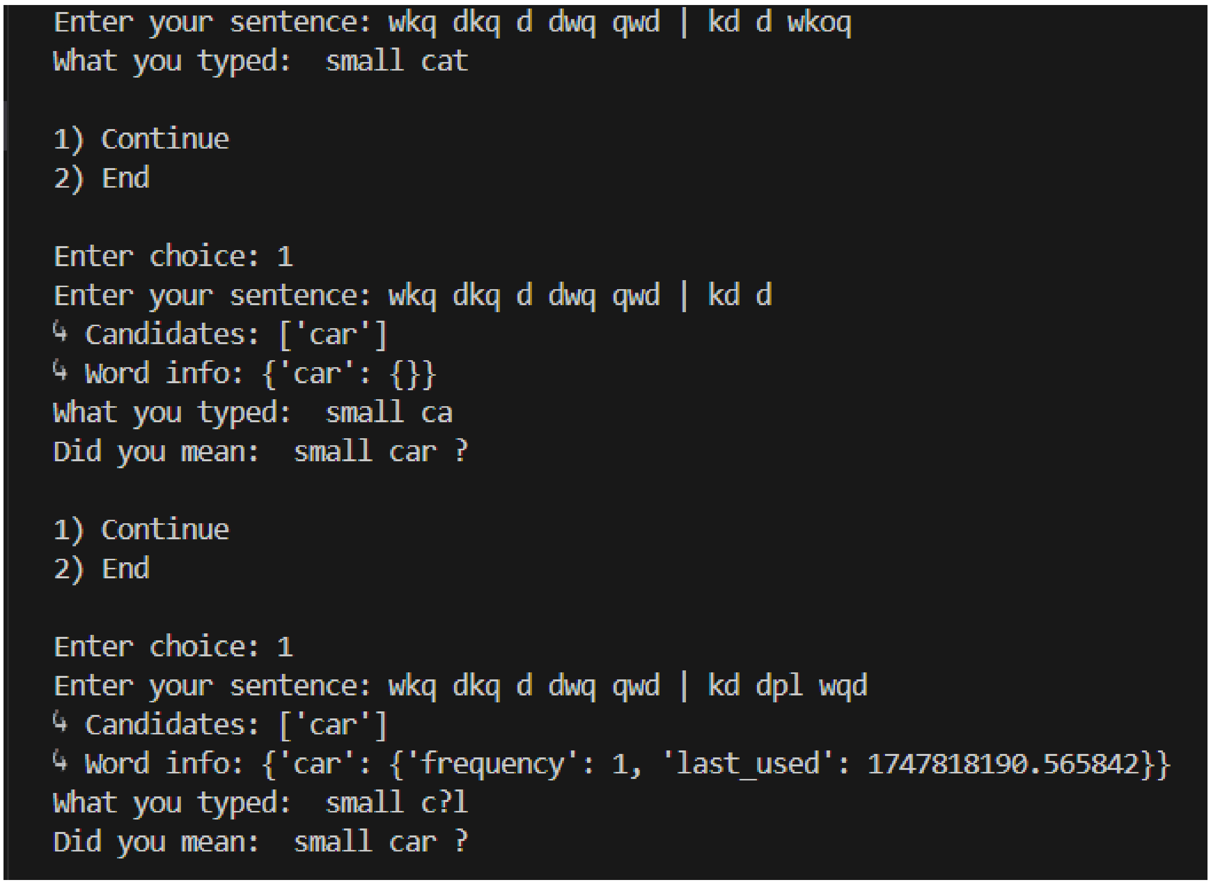
This version took a huge leap in terms of core algorithm and final suggestion logic. Something that bothered me in the 1st version was the fact that I was implementing a simple autocorrect system and it was not altered for or subjected towards improving typing for blind people specifically-who this project is made for.

Upon some light research I found out that new braille typing users make more keystroke-based errors along with spelling errors as compared to intermediate or expert users who tend to make only more spelling errors- just like any general keyboard user.

I wanted to take an approach for solving that issue as well and hence came up with the idea of applying weights to my Levenshtein implementation. Firstly, I added a new dictionary to my 'qwerty_to_braille' module called 'dot_values', which was a collection of dot values in a binary form for each alphabet- representing their key stroke.

Hence, the idea is- once the user types their input, it is converted to their respective English words(combination of English letters they typed in) and when compared to a word in the dictionary, each character's respective keystrokes are compared and every mismatched keystrokes is summed up and added to the Levenshtein distance for that cell instead of the regular 1 added. This way if a greater number of dots (keystrokes) are mismatched then more distance is added- thus lowering the word's chance of being the best replacement. And so, I decided to continue using a Levenshtein algorithm over a Trie tree so that this weight feature could be implemented. For optimizing the code: only words with a length difference less than 4 were processed.

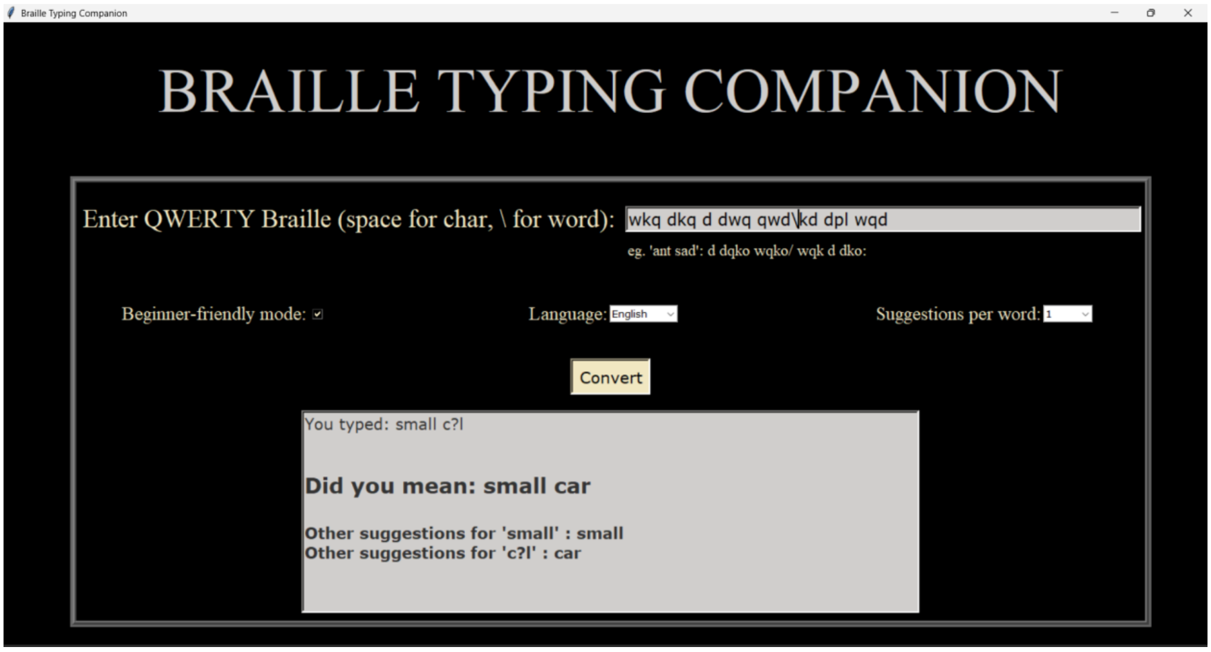
Incase several words have the same Levenshtein distance- the most frequently used or recurring word and then the most recently used out of all is chosen and recommended. This is done by maintaining a dictionary called 'word_info' which holds all the required information of a previously used word during the runtime of the program.



```
Enter your sentence: wkq dkq d dwq qwd | kd d wkoq
What you typed:  small cat

1) Continue
2) End

Enter choice: 1
Enter your sentence: wkq dkq d dwq qwd | kd d
↳ Candidates: ['car']
↳ Word info: {'car': {}}
What you typed:  small ca
Did you mean:  small car ?

1) Continue
2) End

Enter choice: 1
Enter your sentence: wkq dkq d dwq qwd | kd dpl wqd
↳ Candidates: ['car']
↳ Word info: {'car': {'frequency': 1, 'last_used': 1747818190.565842}}
What you typed:  small c?l
Did you mean:  small car ?
```

This version was still missing a spark of specialty and other features to improve accessibility.

Final Version:



Enter QWERTY Braille (space for char, \ for word): wkq dkq d dwq qwd\kd dpl wqd

eg. 'ant sad': d dqko wqko/ wqk d dko:

Beginner-friendly mode: ✓ Language: English Suggestions per word: 1

```
You typed: small c?l

Did you mean: small car

Other suggestions for 'small' : small
Other suggestions for 'c?l' : car
```

The final version of the prototype included most of the features of Version 2 and added several other add-ons to potentially improve the user experience.

Here are all the final features:

- **Main algorithm**: The core algorithm uses Levenshtein distance for calculating the shortest distance.
- **Dot-distance**: For better tackling the common issues faced by any QWERTY keyboard to braille users a weighted Levenshtein aided by a 'dot_distance' module is used.
  How it works: a 'dot_distance' module which maps each alphabet to its corresponding dots (represented in binary form) is stored in the 'qwerty_to_braille' file. If the user chooses 'beginner friendly mode' then the weighted Levenshtein is activated, and the weight is calculated by adding the mismatched dots between the typed and dictionary word. This way the distance between words with lesser key stroke differences will be lower. Thus, increasing its chance to be a replacement.

- **Beginner-friendly mode**: A weighted Levenshtein is added specifically for users who are new to the QWERTY to braille typing system. These users are more prone to causing keystroke mistakes than expert or intermediate typers who are more likely to cause spelling errors. Therefore, this is a 'beginner-friendly mode' option, the user can choose to include or remove for higher optimization- in case an expert typer doesn't need to compare dots errors. This system mimics the real-world typos and errors faced by a visually impaired learner and aims to solve it.
- **Multilingual**: Allows the usage of 2 language vocabularies: English and Spanish
- **GUI interface**: It consists of a GUI with an entry frame, output frame and allows the users to modify their companion by choosing to opt for beginner-friendly mode, language of choice and number of suggestions per word
- **Narrator**: Also consists of a narrator tool which gives a short intro upon opening the page and reads out the final corrected sentence- upon hearing which the users can move to the next word or find the right word from the given suggestion
- **Optimization**: steps taken for achieving optimization are:-
  - **Length based check**: The program doesn't consider words from the dictionary which has a distance greater than 4 units from the typed in words
  - **Optional beginner mode**: Allows expert and intermediate users to choose not to waste time in addressing typos caused by keystroke errors
- **Learning Mechanisms**: The frequency of the number of times a word is used and the time a word was last used during runtime is stored in the 'word_info' dictionary. This information is used along with the

Levenshtein distance to provide the best possible word suggestions. Suggestions prioritize the lowest Levenshtein distance, highest frequency, and most recent use to resolve ties.

Potential improvements for future versions:-

The GUI along with other portions of the algorithm is merely a prototype and can be immensely improved upon becoming a more user-friendly version of the software.

- A model can be added to handle adjacent key errors for characters outside the valid QWERTY Braille keys (d, w, q, k, o, p)
- Optimization can be further improved by adding a Trie tree data structure on final suggestion or preprocessing the word dictionaries to store their info as a BK-tree with the edge weight as the Levenshtein distance between the words in 2 adjacent nodes.
- Other features like including Grade 2 braille typing features could be included
- Real-time inputs can be implemented instead of the current input format with backslashes for space and space between characters.
- The narrator can also immediately provide suggestions in real-time, allowing the user to immediately make the required adjustments.
- Converting every feature to be fully user-friendly including choosing the user mode, number of suggestions and language.
- A Spanish specific narrator could be implemented too- which cannot currently be done because it would require the system setting to be tuned specific to the narrator requirements which would differ from system to system.