Substitution Cipher Report

I chose this method because I wanted to try something different other than frequency analysis. After researching on the topic I couldn't figure out a good concrete way of solving the problem using frequency analysis, so I decided to change my way of thinking. I started to think of what the encrypted text words have in common with English words. Their length. Now I just needed to be able to find a way to map these words. By converting the ciphertext words and English dictionary words to numbers I could then start to map them together using a dictionary mapping. I looked through all the potential words that an encrypted word could map to and I was able to map each encrypted letter to the English letter based off these words. Once I got this far and designed the necessary functions I was stuck. Through various efforts of trial and error I couldn’t figure out how to then reduce each mapping down to only one letter. Luckily, through research on the internet I came across a website called "inventwithpython.com". This explained how I could progress. They had a couple of useful functions in which I used in order to help break down the mapping of encrypted letters to one English letter. A function which got the common letters of two dictionaries and mapped them to a new dictionary and a function that removes “solved” potential decryption letters (encrypted letters that map to only one English letter) from the other encrypted letters’ lists. After this, I was able to then form the key and finally decrypt the encrypted text.

My program relied on one function passing into the next so there wasn't much I could use for threading or multiprocessing. I first tried use threading on the make\_dictionary\_word\_patterns() and get\_final\_mapping() functions. I realised that in order to use the get\_final\_mapping() function I would need to implement a queue in order to store the result of the thread which could then be passed into other functions later in the program. I started without using .join() on both functions. I used the substitution cipher “Russell-cipher.txt” to test initially (about 69 words). This sped up the program considerably, reducing the time on average from 0.2 seconds(without threading) to 0.07 seconds(with threading). But there was one major flaw with this, while the program was creating that dictionary, the rest of the program continued and so the dictionary wasn't fully made by the time the other functions needed it. This resulted in the program not finding any of the letters and returning an empty key and non decrypted text. I tried to slow down the other functions using time.sleep() but this had no effect, I got the same result. I realised I needed to use .join(). The main purpose of .join() is to ensure that a child process has completed before the main process does anything that depends on the work of the child process. This blocks the rest of the main() function from running until the make\_dictionary\_number\_pattern() function finishes. I got the same result as the non threaded program, running on average at 0.2 seconds. I tried using multi-processes on the same function but it didn't speed up and function and didn't decrypt the text either. I also tested it with the Caesar-cipher “two-cities-caesar-cipher.txt” which produced similar results, decrypting the text and finding the key within 0.46 seconds. This text file was longer (about 200 words) then the previous one tested so I therefore found out that the longer the text file, the longer it would take to crack the cipher. I realised I now had to test on a larger scale. I tested a file (about 1500 words) with the same threading in place and it took 2.5 seconds on average to decrypt. It got a lot of the key and was able to decrypt most of the encrypted text. I took away the threads and tested it with the original version(non threaded) and I saw some interesting results. It was now taking around 3 seconds to decrypt on average. After numerous attempts with threading and non-threading it became clear that the threaded version was working faster than the non-threaded version. For my last test, I tested a file(about 9000 words). With this, I found my best results. I had the thread working on the make\_dictionary\_number\_patterns and get\_final\_mapping() functions and .join() was also being used. I was getting it to run between 18-20 seconds. I then decided to remove .join() just to see if there was any effect now that we have a large text-file. It completed in 15-17 seconds. At most saving 5 seconds and at least saving 1 second. As the text-file was very large, I was able to avoid using . join() therefore allowing the dictionary of English words to form and the program to work as expected. I also tested the non-threaded version with this same text file and I got the same results as I did when I used .join() (18-20 seconds). As I don’t know what text-files will be used to grade this project I will leave .join() in my code but I am aware that this isn’t as efficient (in some cases) as not using it. I have come to the conclusion that the threading has sped up the program. I also tried implementing multiprocessing on this same function but it didn’t work just like before.

I think I got the results I did because initially with a small cipher text-file there wasn’t much work for the program so using threading or not using threading had no big difference. This is why it ran it in around the same amount of time. The larger the cipher-text the clearer it became. This is because there was a lot more work for the program. It was able to run the thread without .join() and still execute the program as expected because it had enough time to create the English dictionary mapping of words to numbers.

If I was to undertake this again, I would try to avoid copious amount of words. My implementation relied on a really long dictionary of words. I assume this slowed down my program and therefore isn't as efficient as it could be. I would also try write my program with multithreading/multiprocessing in mind from the start. I think I jumped into a single threaded version and therefore I couldn’t implement multithreading as effective as I could have. Although I learned a huge amount from using inventwithpython.com, I would have liked if I was able to try to get those couple of functions working myself.