CA216 Substitution Cypher Report

By Marius-Constantin Senchea, SID: 18719785, Lecturer: Donal Fitzpatrick

# How and why you chose the method you did to solve this problem?

The method I used is known as the “'hill-climbing” method of solving a simple substitution cypher. I found this method of solving on the [Practical Cryptography](http://practicalcryptography.com/cryptanalysis/stochastic-searching/cryptanalysis-simple-substitution-cipher/) webpage. I also used their quadgrams.txt file together their ngram\_score.py to generate my quadgram\_scores.txt file, which lists all said quadgrams along with their total scores.

I chose this method as, after making several attempts to do a decrypter from scratch (those of which can be found in the other versions/OLD dir), I realised that I was nowhere near as cryptographically inclined as I once thought I was. These old versions include me attempting to use letter frequencies along with the most common 2-4 letter words in English to try and find key pairings. To put it short, it was a total failure. The hill climbing method is also easily understandable and seemed relatively simple to implement, it was also clear to me how I could improve it using multiprocessing.

The basics of the hill-climbing method are as follows:

You start off with a random key. You swap two random letters of the key. You apply this key to your text, and then you score the quadgrams of the decrypted text. You remember the best score attained and swap another two random letters. If the new key is worse than your best key, you revert back to your best key. You do this until you stop seeing a score improvement for a defined number of iterations (I picked 1000, but it could be any number). With some luck and chance, you have your key.

However, this method does have its own drawbacks. Firstly, there is a phenomenon known as a “local maximum”, which means your program may be stuck on a false peak of the hill. This will lead to an incorrect key output. Another drawback is that the completion duration of the program is inconsistent, sometimes it may arrive at its peak faster than other runs as it has just gotten luckier with letter swaps.

To combat the local maximum drawback, I made each hill climb (defined in my code as ‘code breaker’) run a total of 9 times and selected the best score and key of each of those 9 runs and used that as the final key. I generated a frequency key, which was the most frequent letters in the cyphertext matched with the most frequent letters in the English language. I used this key for 3 of the 9 runs. The other keys I used were the alphabet in correct order, and the alphabet in a randomly shuffled order. This would hopefully have given the program enough of a spread of keys to try and avoid all of them from hitting a false peak. My method was not perfect though, and due to the random nature of the program, an incorrect key sometimes slips through the cracks. I am aware of a method/calculation known as “simulated annealing” which would prevent this from happening, but I did not have enough time to implement it.

# How you tested your program to determine which method was faster?

I tested my programs on my laptop which has an Intel Core i7-8550U processor and 16 GB of RAM. I do not have any other devices to test my program with so my results may only be reproducible on my own machine.

I was constantly testing my program while writing it. I imported python’s time module in order to time the duration of the program. I had a total of four test files, three of which included the provided os-sub-cipher.txt, Russell-cipher.txt and two-citites-cipher.txt files. I added my own file, alice-in-wonderland-cipher.txt which contained the first chapter of Alice in Wonderland which was found copyright-free on the web and encrypted using pycipher. I then made a bash script (which can be found in the tests dir) in order to test both my single and multiprocessing versions of my deciphering script against all four text files. I ran the tests 20 times for each file, and I outputted each of their times into their own results files (which can be found in tests/results). I originally wanted to test each version of the decipher script on each test file 100 times to have a nicer average, however I ended up reducing it to 20 as it would have taken at just over an hour to run just the single-process version of the script on the os-sub-cipher.txt file (meaning the whole test suite would have taken at least 4 hours to run). I then wrote a short python script to calculate the averages of each of the results files and output them all into one file, all\_averages.txt.

# Which method was faster?

I have attempted to make the following table as accessible as possible as per Microsoft’s guidelines. Apologies if it is not good enough.

Table 1 Single Versus Multiprocessing Comparison

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Single-process average run time** | **Multi-process average run time** | **time decrease percentage** |
| **Russell-cipher** | 2.62 seconds | 1.16 seconds | 55.73% |
| **Alice-in-wonderland-cipher** | 54.89 seconds | 22.12 seconds | 59.7% |
| **Os-sub-cipher** | 57.05 seconds | 24.2 seconds | 57.58% |
| **Two-cities-cipher** | 6.4 seconds | 2.79 seconds | 56.4% |

As shown in the table, the multi-process implementations reduced the running duration of the script by at least 50% for each test file. There is a clear benefit to running my script which implements multiprocessing, and it is definitely the faster of the two implementations. I originally had another version of the decipher script which also split the text input into chunks and assigned threads/processes (I tried both) to count their letter frequencies. There was no added benefit to doing this, as it took the same amount of time to complete as my final program did. I assume that this was due to the amount of time it takes for threads/processes to start up, and it may be better suited for even longer bodies of text than what I was using.

# Explain your results. Why do you think you got the answers you did?

As stated before, adding multiprocessing gave my program a running time decrease of at least 50% per text file. I believe I got these results due to the way my program is written. Instead of having to code break 9 times in a row, my program does a few of the code breaks at about the same time. This speeds up the process by quite a bit. These results were about what I would have expected.

# What would you do differently were you to undertake this exercise again?

I would look into simulated annealing and try to implement that, as it may be even faster than my method, and it would definitely be more accurate. I would also try to read more into other methods of cracking the sub cypher, as I may not have found the best method out there.