CA216 Cracking the Substitution Cipher

I have chosen the single-threaded program as the most efficient.

Disclaimer

The OS I am using is Windows 10. I am unable to test my code against a Unix based system so it may not be the best in a Unix environment.

How and why I chose this approach:

The approach I took towards this project is an attempt at using the hill-climbing and simulated annealing algorithms.

At first I was unsure on how to begin so I opted for other ways such as assigning the letter “e” to the most common letter hence assigning the most

popular three letter word to the word “the” and so on. At the beginning this approach somewhat worked but I found myself basically unknowingly tailoring

my program to suit the sample text files. It also ended up to be at least around 400 lines, which was the point that I realised this approach might be a “little”

too hardcoded. I then realised it was a futile attempt and looked for much more efficient ways. I came across the simulated annealing and hill-climbing algorithms

but I found it quite daunting due to the seemingly complicated mathematical logic of it all. At that point I even wanted to give up and solve the caesar cipher instead.

However I did more research and found the reasoning behind it was actually quite simple, it just involved a lot of statistics and logarithms.

I chose this approach as it has been the most effective and fastest out of all of my other attempts.

Results of my single threading, multithreading and multiprocessing tests

The only change I made between the three versions is the placing of the main loop/algorithm. I found that I could not change other parts of the code as most of it needed to be run in chronological order.

I imported a python module called “time”. I used its “perf\_counter” function to determine the running time of the program. For example, I had a variable at the beginning and end of the code called “start” and

“finish” respectively, which each contained the value “time.perf\_counter()” . At the end of the code I printed the value of “finish - start”. I first came across this idea when I watched this video on

multithreading:

https://www.youtube.com/watch?v=IEEhzQoKtQU

For each test, I also checked its accuracy by monitoring manually how the decrypted files changed with each newly generated key. I ran this time and accuracy test for each file 10 times, for each version of my code

(single-threaded,multiprocessing, and multithreading,mixed). Although the results shown here are calculated averages, I found these tests occasionally gave me fluctuating numbers. This depends on how many threads/processes

are currently running in the background.

SINGLE THREADING RESULTS

- russell-cipher.txt : 9.334 seconds

- two-cities-cipher.txt : 15.684 seconds

- os-sub-cipher.txt : 102.301 seconds

MULTITHREADING RESULTS

To test the program’s result with multithreading I used the module “concurrent.futures”. This is simply because I needed the return value of the thread and I found it was the simplest way to do so.

First I moved the main loop into a function called “loop” and ran the inner while loop for the same minimum amount of times as the single threaded version. I used two concurrent threads and compared

their scores to choose the best key. These are the results I got:

- russell-cipher.txt : 5.936 seconds

- two-cities-cipher.txt : 13.648 seconds

- os-sub-cipher.txt : 121.747 seconds

I then tried halving and dividing the number of iterations of the for loop in the “loop” function in an attempt to make it a fair comparison. I found that the results were more or less still the same.

MULTIPROCESSING RESULTS

I also used the “concurrent.futures” module for the multi-processing version and used the same tests as the ones in the multithreading, where I used 2 threads.

These are the results:

- russell-cipher.txt : 6.350 seconds

- two-cities-cipher.txt : 14.613 seconds

- os-sub-cipher.txt : 127.730 seconds

I also tried the halving and dividing tests on this code and found that yet again it produced similar results still.

MULTIPROCESSING AND MULTITHREADING RESULTS

I ran the same tests as before but had one thread and one process. Again, it did not show any major change in timing.

Explaining my Results

Both my multiprocessing and multithreading test results are much faster than the single-threaded results in terms of the shorter text ie “russell-cipher.txt” and “two-cities-cipher.txt”.

However for large text files such as “os-sub-cipher.txt” I found that the single threaded version proved to be much more efficient. It ran for approximately 60 seconds whilst the others ran for around 120 seconds.

I believe this is due to the fact that I am running this code on Windows. I have learned that efficient multithreading and multiprocessing with demanding programs is actually quite difficult to achieve with python let alone this operating system.

As I mentioned before, the run time of each test does differ quite a lot due to background processes and threads. This is because Windows has an issue where it has trouble handling over 64 threads at the same time. When the thread count goes above this number,

it separates these threads into different processor groups. For example the first batch of 64 go into the first group, the second batch go into the second group etc. From my understanding, this means that instead of a multithreading approach, the results will more than likely be similar to that of a multiprocessing one.

WIndows multiprocessing and Unix multiprocessing differ greatly in performance. Windows uses a “spawn” method whereas Unix uses a “fork” method. While in the “fork” method, the parent process and the child processes will be completely identical,

which is not the case for the “spawn” method. Instead the child process will only be given the necessary resources to run.

I decided to choose the single threaded version as the most efficient as there is not much of a difference between their results in terms of short text, but there is an obvious gap in time as the text gets larger in length,

assuming there are multiple processes/threads running in the background. I believe this might also be due to the fact that the overhead cost of moving the data between the processes is quite large.

What I would do differently if I were to undertake this task again

I would definitely try to recreate this task in a Unix-based Operating system and I would like to figure out a way to make the program run faster.

I would try to implement this cipher decoder in a different language that is able to run concurrently. I would be interested in how much more accurate the results could be even when run for a shorter

amount of time, but with multiple threads and processes.

RESOURCES

Probability logic:

http://practicalcryptography.com/cryptanalysis/text-characterisation/quadgrams/

http://practicalcryptography.com/cryptanalysis/stochastic-searching/cryptanalysis-simple-substitution-cipher/

Hill Climbing Algorithm

https://wildonblog.wordpress.com/2018/05/22/hill-climbing-on-substitution-ciphers/

Simulated Annealing

https://www.r-bloggers.com/decoding-a-substitution-cipher-using-simulated-annealing/

https://www.tandfonline.com/doi/abs/10.1080/0161-119391868033

Multiprocessing/threading in Python

https://hackernoon.com/concurrent-programming-in-python-is-not-what-you-think-it-is-b6439c3f3e6a

Multithreading in Windows

https://www.anandtech.com/show/15483/amd-threadripper-3990x-review/3