**Andrew Finn**

**18XXXXXX**

[**andrew.finn8@mail.dcu.ie**](mailto:andrew.finn8@mail.dcu.ie)

**Cracking the Substitution Cipher**

***How do you use your program?***

Python3 sub\_decrypt.py [FILENAME]

If no file name is specified program will take input via command line after starting.

Functions can disabled / modified by changing values on the first lines of the program as outlined in the inline comments. Values should be changed when using processes as outlined in the program. If the output is slightly off this can be resolved by raising the correct percent variable and increasing the freezing iteration variable. When using a large number of processes count the freeze index can be slightly reduced due to the score of each key being higher.

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

After trying and failing on several occasions to solve the problem using frequency analysis, I went back to drawing board. After several more attempts I found information and code examples of hill climbing method however this implantation didn’t work due to the program in going down the incorrect ‘path’ much of the time. After more research I found annealing simulation in, this approach uses the hill climbing method but cools the temperature over time to reach the global maximum instead of one of the many local maxima, However in solution this sometimes fails, to deal with this I check to see if progress is made and if it hasn’t for X amount of iterations I restart and try again. In my implementation I ask to the user to check the output after a predefined percentage matches words in a hash set to unsure that in fact the global maximum was found and starting again if otherwise. This approach works due the class NGramScore which is popular in solving a substitution cipher, this class used a set of monograms, trigrams , quad-grams etc. to calculate a log probability of each occurrence and in turn score a key and text which can then be randomly changed and this series is repeated until the correct text is found.

***How you tested your program to determine which method was faster. In other words, is the single threaded (as in perfectly normal/traditional) implementation faster than one using multiprocessing, threading, or a combination of both.***

Testing was undertaken by running each test file 10 times with 0, 1, 4, 12 processes respectively, disregarding attempts where the program doesn’t get the correct answer the first time the number is then averaged to get a rough approximation. This is all that is needed for my solution due to the fact that my solution works based on randomness and as such accurate times/consistent times are never going to be achieved.

The results did confirm that single threaded for my application is the quickest followed by 12, 4, 1 in that order. With no processes the solution was found in less than 2 seconds on average(1.2018 seconds), less than 6 seconds on 12(5.4037 seconds) and 4(5.8208 seconds) processes varying significantly between runs due to the randomness and greater than 30 seconds on a single process(46.7057). There is a high variation of single process due to a high iteration count resulting in a large amount of processes being created on some runs.

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

Running small test files repeatedly I average around .5 – 2 seconds to solve for the solution. For the purposes of testing I disregarded outliers when the program required to restart after choosing an incorrect ‘path’. When running with 1 process the program is significantly slower taking 5 to 10 times more than single threaded due to the fact a process must be initialized each iteration and only runs the same number of keys as single threaded each iteration, i.e. The program is doing the exact same thing but starting a process every iteration. I am running a 12-core processor I allowed the program to use 12 processes, this ran on average in 15-30 seconds. While slower than single threaded the program solved the cipher in significantly less iterations. The same was through for 4 processes running a few seconds slower than 12 processes but doing 1/3 the work.

The issue with my solution is each step require output from the previous step meaning processes would be waiting on the previous output defeating the benefit. As such the best solution I could come up with was to run several iterations each iteration and using the best one to proceed. The issue with this is the overhead associated with starting new processes, this mitigates any potential gain and instead runs slower than running single threaded. As such each additional core has a lesser or equal effect than the previous core. Similar to the economic law of diminishing returns, a doubling in processes does not create a doubling in performance. In some cases, with large process count more processes could even slow the program down.

No, I did not expect the results to be as they are, I was aware there was an overhead to using processes/threading in Python however I was not aware just how significant this overhead is. I can’t speak for other approached but for mine it is not suitable to be multithreaded unless you were trying to run several different text files the same time. Threading and processes in my opinion is most circumstances is suited for running separate events rather than to speed up one. This however isn’t always the case if a program is designed and suited for threading/processes it could be designed that each section would be independent of all others and instead a final function would combine to create the final answer. This can be seen when running the program with 1 process and single threaded side be side. Each program is doing almost the exact same work but due to the overhead from processes is significantly slower.

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

Use a different method to solve the problem. The issue/benefit with annealing simulation is its speed. Running single threaded in less than 2 seconds made it a solution that wouldn’t significantly benefit from multi-processing. One solution to this would be using a less efficient method and designed from the ground up using threading.