**Cracking the Substitution Cipher**

**Solution:**

Initially when I started this project my solution consisted on rudimentary frequency analysis. I counted each individual letter, divided it by the total number of letters and multiplied by 100. This gave me a rough percentage that I then compared to the frequencies found on The Black Chamber.

With this solution I was successfully decoding 3 to 4 letters, those being e, t, a and o. This however, was dependent on the length of the encrypted message. Anything less than 60 to 70 characters would significantly reduce the success rate of the program. I suspect that any character that was found correct given such a small input file was because of chance.

After some more tweaking and implementing a system to find the characters ‘a’ and ‘i’ manually by looking at single letter words I decided that I needed to change my approach. I found this solution: <https://inventwithpython.com/hacking/chapter18.html>

It involves creating word patterns for the encrypted message and then compering them to the word patterns of English words from a dictionary. A word pattern is an incremental numerical assignment of each letter in a given word. For example the word PUPPY has the following word pattern: 0.1.0.0.2, the word PLANE has 0.1.2.3.4 as its respective word pattern, the word MOON has 0.1.1.2 as its respective word pattern, etc.

An encrypted word will have the same word pattern as its decoded counterpart. So DRDDM and PUPPY both have 0.1.0.0.2 as their word pattern. However, this technique to break the substitution cipher merely reduces the range of permutations of an encrypted word. Both the word PUPPY and MUMMY have the same word pattern and as such an encrypted word like DRDDM could be either of those. This solution requires additional analysis in order to deduce the correct key. It is also reliant on the quality of the dictionary provided to construct its word pattern data set. As such I deemed the technique inadequate and decided to start over again.

After some more research I came across the hill climbing algorithm and its potential application to crack simple substitution ciphers. I used the technique described by the following web page:

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

The hill climbing algorithm belongs in the genetic algorithm family. They are primarily used in optimization problems. Genetic algorithms try to mimic the natural selection process that happens in nature. A simplified overview would be: generate a solution, score it, generate another solution based on the original, score it as well, compare the two scores and keep the better solution. Repeat the process until you achieve a good enough solution.

All genetic algorithms have a way to score their solutions. In English, quadgrams, which are groupings of 4 letters, similarly to individual letters have distinct frequencies. Some quadgrams occur more than others. I used the steps outlined here:

[http://practicalcryptography.com/cryptanalysis/text-characterisation/quadgrams/#a-python-implementation](http://practicalcryptography.com/cryptanalysis/text-characterisation/quadgrams/" \l "a-python-implementation)

to write my own program, quadgrams\_score.py, to calculate the log probability of English quadgrams and load them into memory as a dictionary where the quadgram is the key and its log probability is the value. I used the file quadgrams.txt provided by the website above as my data set. It contains a list of quadgrams and their count from English texts. The formula for the log probability is quite simple: log(count of the quadgram divided by the total count of quadgrams). Logs are used because when you multiply small floating points numerical underflow can occur.

The main program first generates a random key as a parent value. It then decodes the encrypted file using this randomly generated key. The output text is then divided into quadgrams. The main program then searches these quadgrams in the dictionary created by quadgrams\_score.py and adds them all together in order to score the decryption key. It goes by reason that an encrypted message will contain quadgrams that have low probabilities as they occur less often in English. If a quadgram does not exist in the quadgrams\_score.py dictionary then 10 is subtracted from the over all score as a penalty.

After scoring the parent, another key is generated known as the child key. The child is a slightly randomized version of the parent where two values are swapped with each other. So for example:

parent = {‘a’: ‘e’, ‘b’: ‘m’, ‘c’: ’p’…}

child = {‘a’: ‘m’, ‘b’: ‘e’, ‘c’: ‘p’…}

The child is then scored in the same way as the parent. The two scores are compared and if the child has a better score than the parent then it becomes the new parent. If it has a worse score then it simply ‘mutates’, i.e: it is slightly randomized again. This processed is repeated 10000 steps/generations. At the end we have a key that is capable of decoding the input file.

One downside of the hill climbing algorithm is its ability to get stuck in what is known as a local maxima. Essentially no improvement can be achieved from randomizing the parent, however the parent itself is not the correct key. To avoid this happening I check to see if the parent has change in the last 1000 generations and if it has not then I randomize it. I also run the main program 4 times to ensure that at least one answer is not a key that got stuck on a local maximum. The number 4 is arbitrary however from testing I have found that any lower number will cause the chance of the key being incorrect to greatly increase.

In the multi process version of the program I assign each rerun to a separate process. This significantly speeds up the decoding time.

**Testing:**

To test both my single and multi process versions of my program I wrote a small python file (test\_sub\_cipher.py) to run the programs 100 times each and record the time it took for each run into a file. The test program also compared the solution key and the key generated by my own encryption program and recorded how many correct characters were decoded. After each run the input file was encrypted again. The results of the tests can be found in single-process-tests.txt and multi-process-tests.txt respectively.

The average time for the single process program was 9.894 seconds and on average decoded correctly 23.58 letters.

The average time for the multi process program was 6.433 seconds and on average decoded correctly 23.72 letters.

Splitting the reruns of the algorithm onto multiple processes resulted in a decrease in the time taken to decode the given file. It also seems that the multi process version of the program was more successful at decoding, finding more correct letters, however this was marginal.

I also would like to add that when running the multi process version only once the decoding process takes only 4.57 seconds. It appears than when running the program repeatedly the decoding process increases to 6.43 seconds. I am unsure why this happens but I suspect that destroying processes is computationally expensive and is slowing things down.

**Reflection:**

If I were to redo this task I would definitely research and try to implement simulated annealing which is another technique that tries to prevent algorithms from getting stuck in a local maxima.

The main slowdown I have right now with my program is the need to rerun things in order to make sure they are correct. With simulated annealing I would expect a significant increase in speed.