Fall 2020 CS313 Prof. Fried

Group 5: Problem no. 6 (Vigenere)

Muhammad Abbas Ali Sajjad.

The Vigenere cipher, nicknamed le chiffre indéchiffrable which means the indecipherable in French, is arguably the most famous cipher after the Caesar cipher. The basis of encryption being, each character is converted to a cipher, letter by letter, following a unique key. The key being any series of characters representing how many positions down the alphabet the original character shifts over. 'A' being a shift of 0 and 'Z' being 25.

Cracking the Vigenere cipher involves 2 steps, the first being, determining the length of the cipher and once you know the length you can begin the process of producing a key. The problem at hand was to decode a key with a length of 32 using a genetic recombination algorithm. This proved to be quite challenging and quite fun.

Our program begins by generating random keys into vectors<type> that meet a certain threshold in "score." The score being as follows. We generate a random key and encode a cipher text using a plaintext. We compare the new cipher text with the cipher text from the real key, character by character, and produce a score, 32 being the highest score, implying a perfect match. Determining an adequate score threshold proved to be difficult, we set our random key generator register_keys() function to produce 25 keys with a score of at least 8. A threshold of 9 or higher took too long.

The next step is to run a series of recombinations and produce successive generations of keys that elicit better scores. Producing a generation of keys, requires 2 parent keys to produce 2 offspring keys. The keys selected for recombination are random. My generation() function has 2 components, a recombination() component and a mutation() component. The recombination finds cross-over points on parent keys and swaps key values between them producing 2 unique offspring keys.

Initially, I began with a fixed cross over point at 50% recombination, meaning that the offspring keys receive 50% from each parent. But after a series of generations my key scores were not increasing, sometimes even decreasing. So I opted for a random cross-over point at 50% which was slightly better but not successful enough. I then chose to have a random 25% multi cross-over recombination, which proved to be most successful. The way a multi cross-over point works is by finding a random initial point and then a subsequent random terminal point and crossing over everything in between.

The second component of the generation() function is the mutation() function. Mutation is a naturally occurring phenomenon in genetics that postulates random changes are healthy for adaptation overtime. The key to a successful mutation is determining a useful probability or mutation rate. If you mutate too aggressively you can damage successive generations lowering their score. If you mutate too passively you make no contribution to future generations.

My program mutated keys at a rate of 0.01 or 1%. Which translates to every 3rd time 2 keys are selected for recombination, 1 of the 2 keys is selected for mutation and a single random character in the key is mutated to a random character.

After producing generations of keys that produce better scores, i.e. return a cipher that matches the original cipher, our next step is to pull out the best keys. We replace our current registry with the best keys from subsequent generations and then run a series of recombinations on them, producing yet another set of keys with even better scores and the cycle continues until a key is found.

The challenging part is choosing parameters and thresholds. It all comes down to running hundreds of subsequent recombinations and tweaking slightly each time to see what works and what doesn't. There are so many variables to consider that the task can become convoluted. There are many questions somewhat left unanswered. How many random keys do you begin with? At what threshold score do you want those random keys? Now that you have random keys how many recombinations do you do? Each time you need to pull out the top performers, but when you run new recombinations on those top performers to create a new generation, what threshold score have you determined as the criteria to move onto the next generation. It is simple to think "as long as scores are improving" but evidently you will run into a plateau where your scores seem to stagnant. The goal is then to view your strongest generation and decipher the key either by human evaluation or run recombinations until the program returns a perfect score.

Another challenge I faced was ensuring that I always get enough keys in the new generation. My algorithm grabbed the best keys as they were produced meeting a certain threshold, but if my constantly increasing threshold was ever too stringent at any point down the line I pulled too few keys resulting in fewer keys to recombine.

Ultimately, cracking Vigenere though a genetic recombination algorithm is definitely possible. You become mesmerized by the technique, as you begin to see your keys starting to take shape, returning higher and higher scores. Just as nature does with genetics since the dawn of time.