KRY project 1

Vigenère cipher

Patrik Németh, xnemet04

Implementation

Here are outlined the methods used for cracking the Vigenère cipher, and how they were implemeted in the program. The methods themselves are not explained in detail, but some context may be added when describing implementation.

Firstly, the program loads the input cipher from the standard input stream into an `std::vector` structure. The loaded ciphertext is converted into uppercase letters, as working with a single case is easier. This does not affect the accuracy of key cracking, because the input ciphertext is monocase. Next, the keylength estimations are calculated and printed in the required order and lastly the key is cracked and printed as well.

Friedman test

The basic variation of the Friedman test uses an empirical constant *Kp* describing the probability of selecting two identical letters from an arbitrary alphabet based on the frequency, at which that letter is normally used in a language using that alphabet. For the English language this constant tends to be around 0.065, and this is the value chosen for this project. The constant *Kr* describes the coincidental probability of the same. For English this value is 0.0385.

The last variable needed for finding the estimate of the key length is K_0, which is the same probability as *Kp*, only calculated for the input text. For its calculation, individual letter frequencies need to be counted, which is done in `getLetterCounts()`. This function returns an array of integers of size 26, where every *i*-th element represents the count of the *i*-th alphabet letter in the ciphertext. Next, `calcIC()` calculates the index of coincidence K_0. The resulting key length estimate is calculated as $(K_p - K_r)/(K_0 - K_r)$.

Kasiski test

This method exploits the fact, that a cipher key is repeated over the entire length of the source text during its encryption. This has the effect of giving a chance for multiple parts of the source text containing the same letter sequences to be encrypted with the same part of the key. Because of this the sequences containing same letters in the source text will be encrypted to sequences of same letters in the cipher. Sequences of matching trigrams in the ciphertext and their respective distances from each other are found by calling `trigramDistances()` and then the commonest trigram is selected. The greatest common divisor is calculated for every combination of distances of this trigram. Of these divisors, the commonest one is selected as the most likely key length. Length of 1 is not taken into account, as this would not constitute a Vigenère cipher.

Index of coincidence

This method is similar to the Friedman test, except that it calculates the index of coincidence for individual key lengths. At each attempted key length (implemented to successively try lengths 2 through 150) the ciphertext is broken up into *key length* number of columns. If the current *key length* is the correct length, this gives us *key length* number of ciphertexts, all encrypted with a Caesar cipher. This would mean, that the index of coincidence of these columns would be similar to that of the English language, meaning that we have found the correct key length.

The main loop for trying individual key lengths is in `ICkeylength()`. For each length letter counts are acquired from the individual columns by `getLetterCounts()` and from these counts the index of coincidence is calculated by `calcIC()`. The average index of coincidence is calculated for the tested columns and if this average is in the interval of <0.062, 0.073>, then the currently attempted key length is accepted as the correct key length. This thresholding was implemented after finding, that multiples of the correct key length tended to produce IC (index of coincidence) values closer to the desired IC value of 0.065.

Frequency analysis

For cracking the letters of the key, frequency analysis was implemented. `findKey()` is in charge of cracking the key. The ciphertext is broken into columns to receive a *key length* number of Caesar ciphers, similarly as previously, and for each column a best guess for the alphabet shift is calculated by `guessShift()`. This function takes the empirical frequencies of letters in the English language and the frequencies of letters in the current column of the ciphertext. It cycles through all possible shifts of the alphabet and selects the shift, which is most likely. This is repeated for every column of the ciphertext, after which the found key is returned.