Project 1: Cryptanalysis of a Class of Ciphers Based on Plaintext Informed Hill-Climbing

Sony Maharjan, Teeya Roberts, Andrew Zitek

Department of Computer Science, Tandon School of Engineering, New York University

CS-GY 6903: Applied Cryptography

Prof. Giovanni Di Crescenzo

March 7, 2021

Project 1: Cryptanalysis of a Class of Ciphers Based on Plaintext Informed Hill-Climbing

Introduction

Our team consists of three members: Sony Maharjan, Teeya Roberts and Andrew Zitek. Andrew compiled information from the project description as well as the textbook and other academic literature to develop this unique approach. He tested and fine-tuned the code to successfully execute a program to meet the project requirements. Teeya and Sony checked this technique to make necessary changes that could have been added/omitted from the program. Sony additionally tested program running time for longer plaintexts per the description of extra credit part 1.

Informal Explanation

When hypothesizing for a plaintext given ciphertext, four configuration parameters considered:

1. Length of key t (guaranteed to be 1-24)
2. Shift values at each index of the key $k0 \ldots kt - 1$ (all values in 0-26 inclusive)
3. Rule(s) about selecting an index from the key, meaning, the scheduler behavior
4. Rule(s) about inserting random characters, also scheduler behavior

It is worth noting that a basic approach would tabulate all conceivable rules for 3 & 4 and test each combination with all possible key lengths and key values. This approach would be inefficient, but given infinite time, would result in success (provided all possible rules for 3 & 4 are guessed). In reality, we cannot rely on guessing all possible rules for 3 & 4, and we have finite time to execute a successful crack.

To improve the basic approach listed above, we compiled a number of statistical approaches:

a) *Index of Coincidence* analysis can identify key lengths that result in valid looking English (Katz & Lindell, 2021, p. 13)

b) For each key index, *Frequency Analysis* can identify shift values that produce plaintext with letter frequency that matches our dictionaries (Katz & Lindell, 2021, p. 8)

c) *Levenshtein distance* can identify the fitness (quality) of plaintext guesses resulting from decryption attempts using key guesses (Navarro, 2001, p. 5)

Using a & b together, we can produce a list of initial key guesses and determine a few that look promising by choosing the ones with the smallest Levenshtein distance from any dictionary value. We then switch gears to local search optimization:

d) *Hill-climbing* can further improve the key guesses we've computed through trial and error (Kaeding, 2019, p. 3)

Whereas a purely stochastic process for hill climbing would simply choose random modifications to the key and observe the fitness of the result, our approach is more sophisticated because for each index of each key guess we tabulated in step b what shift values are most probable. We can therefore check only the most promising looking modification and avoid large amounts of unproductive computation.

In the end, if the fitness of some plaintext guess when compared to any dictionary 1 value is less than 200, we interpret this result as a success for test 1 and return the corresponding plaintext. If the distance is greater than 200 then we move on to test 2. In test 2, all procedures are the same except for the fitness function, which is slightly modified.

In test 2, ciphertexts are generated from some composition of, but not necessarily all, entries from the dictionary. Therefore, we wanted to design a simple fitness test that rewards plaintext guesses containing words from dictionary 2 but does not punish guesses when they do not contain a word in dictionary 2. To achieve this, we use a string includes function and assign a distance value of -1 when a word is present, indicating good fitness, otherwise zero indicating neutral fitness. The fitness of a plaintext guess is assessed as a sum of these values over all words in dictionary 2. In the end, we return the plaintext guess with the lowest best fitness per this measure. Of all processes implemented, this fitness function probably has the most room for improvement.

Rigorous Description

Index of Coincidence

Per the book (Katz & Lindell, 2021, p. 15), the index of coincidence method is a methodical and easy-to-automate method for finding key lengths. It approximates the occurrence of two of the same letters appearing next two each other in a text. Different languages have different values for their index of coincidence and the index of coincidence for English text is about 0.65. Basically, the approximation works with nested for-loops as so:

```
for i = 1 … 24 // every possible key length
    for j = 0 … length of ciphertext
        count the occurrence of each letter including space
    SUM k = 0 … 25 count-of-char-k * (count-of-char-k - 1)
    return SUM * (1 / (N * N-1)) // where N is total number of chars
```

The key length with value closest to 0.65 is most likely to be the correct key length.

Frequency Analysis

Per the book (Katz & Lindell, 2021, p. 8), the frequency distribution of individual letters of English language text (and other languages) is known. For example, the letter ‘q’ occurs with far less frequency than the letter ‘e’ in English language text. We have taken this a step further by computing the occurrence of letters in our dictionaries. Basically, it’s done with nested for-loops like so:

```
for i = 0 … 24 // alternatively, only some subset of the most promising key lengths
    j = 0
    while j = 0 <= length of cipher text
        count the occurrence of each letter including space
        j += keyLen
    for k = 0 … 26 // basically “slide” the “fingerprint” over the expected distribution
```

```
        sumOfDist = 0
        for m = 0 … 26 // each letter
            freq-m = divide the occurrences by total letters
            dist-m = Absolute Value (freq-m – actual-freq-m)
            sumOfDist += dist-m
        return the k (or shift value) that minimizes computed freq. from expected freq.
```

Levenshtein Distance

Levenshtein distance is a number that defines how unique two strings are. An example of this is the Levenshtein distance between "mitten" and "fitting", is 3 since, at minimum, there are 3 edits required to change on to the other. An edit can be defined as an insertion, deletion, or replacement of a character. We did not write this function we copy/ pasted from the Github project below.

https://gist.github.com/TheRayTracer/2644387

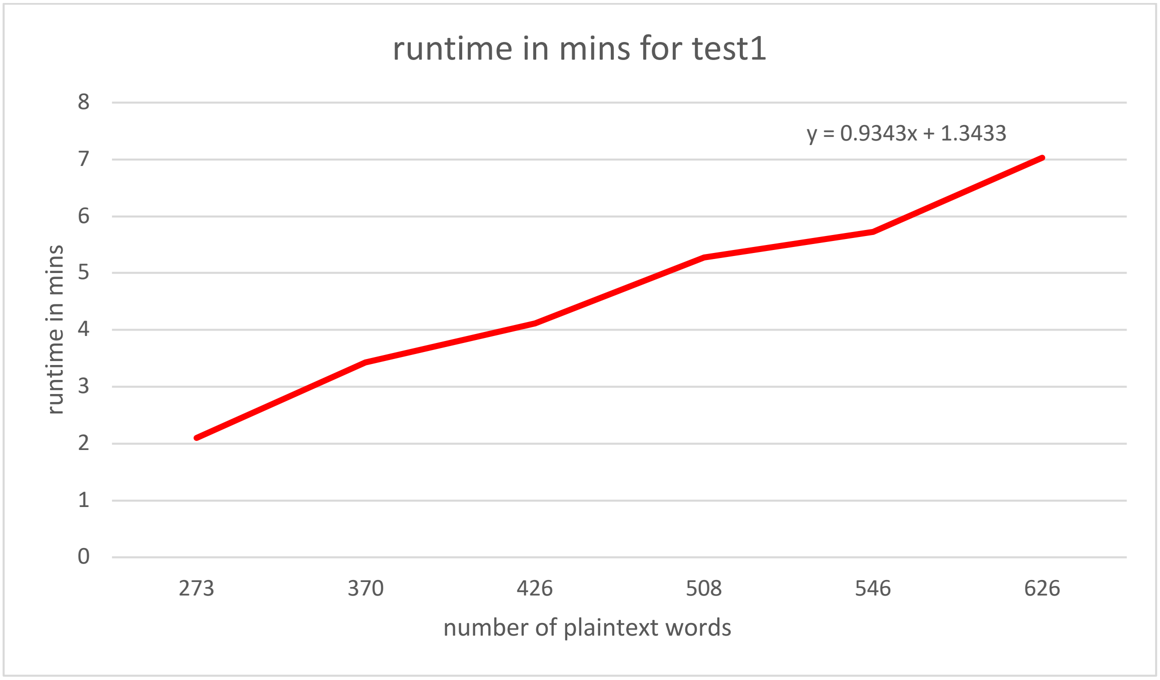
Hill Climbing

Hill climbing is a process of making an edit, checking fitness, and keeping the edit only when fitness has improved. Fitness in this case is Levenshtein distance from a dictionary value. Though this process is often purely stochastic, meaning that the edits are chosen at random, our process instead uses indicators from frequency analysis to make only edits that seem more likely to result in better fitness. This helps to avoid unproductive computation. The process is outlined as follows:

```
for i … SOME NUMBER OF ROUNDS
    newKeyGuess = modify(keyGuess)
    fit = compute_fitness(ciphertext, newKeyGuess)
    if fit > compute_fitness(ciphertext, keyGuess)
        keyGuess = newKeyGuess
    else // nothing. proceed with another modification
```

Extra Credit 1:

For extra credit, we increased the number of plaintexts words *dict1* array in *dictionaries.cc* for test1 and re-ran the program for test1 multiple times. The strategy worked well for additional tests and the runtime calculated in minutes showed a gradual increase in each test. Below we have included a plot chart and the function for runtime vs plaintext words.



References

Katz, J., & Lindell, Y. (2021). Introduction to modern cryptography. CRC Press. https://doi.org/10.1201/9781351133036

Navarro, G. (2001). A guided tour to approximate string matching. ACM Computer Survey. 31–88. https://doi.org/10.1145/375360.375365

Kaeding, T. (2019). Slippery hill-climbing technique for ciphertext-only cryptanalysis of periodic polyalphabetic substitution ciphers. Cryptologia. 44. 1-18. 10.1080/01611194.2019.1655504.