Decrypting Noisy Substitution Ciphertexts

CS 221 Project Progress Report

Brendon Go
bgo@stanford.edu

Colin Man
colinman@stanford.edu

Spencer Yee
spencery@stanford.ed

I The Problem

A substitution cipher is a way of encoding plaintext into a ciphertext. In a simple substitution cipher, first a substitution alphabet is created. The letters of the alphabet is mapped one-to-one with a (not necessarily different) letter. For example, "ABCDEFJHIJKLMNOPQRSTUVWXYZ" can be mapped to "CVNXKFGPMOQBWYUHRIZALTEJSD". Then a message like "THIS IS AN EXAMPLE OF A PLAIN TEXT MESSAGE" can be encrypted with the substitution alphabet into "APMZ MZ CY KJCWHBK UF C HBCMY AKJA WKZZCGK" ('T' maps to 'A', 'H' maps to 'P', etc.). To decrypt the message, one needs to supply the reverse mapping of the substitution alphabet to the regular alphabet.

If there is 'noise' in transmitting a message, each letter has a chance to be accidentally lost, or changed to another letter. There is also a chance that a letter is received when it was not there originally. That is if the sender intended to send the ciphertext "APMZ MZ CY KJCWHBK UF C HBCMY AKJA WKZZCGK" but the channel used to send it was noisy, the receiver may instead have received "AFMZ MM CY KJWHK UF CZ HBAMY KJA WKZCLK". When decoding this message with the substitution alphabet, one gets "TFIS II AN EXMPE OF AS PLTIN EXT MESAUE", clearly not the original message.

II Model and Algorithm

2.1 General Strategy

We chose to model this task as a search problem and to solve it with A* search. A state is represented by a mapping from letters to letters. For example, the state {'A' : 'B', 'M' : 'C', 'K' : 'E'} means we have decided that the letter 'A' in our ciphertext should be decrypted to a B', 'M' should be 'C', and 'K' should be 'E'. Therefore, our starting state is simply the empty mapping, {}. An action is simply a mapping from a single letter to another letter, such as {'B' : 'G'}. Taking action a at state s results in a state $s \cup a$. A state is our goal if it has a mapping for all 26 letters in the alphabet, or at least for all the letters that appear in the ciphertext. For a given state, the available actions are all the possible mappings for the first letter that appears in the ciphertext (from left to right) that does not yet have a mapping.

The cost of an action at a state depends on various factors. So far, we calculate several individual components to determine the overall cost. We begin with an idea of what letter frequencies should be, taken from Cornell's math department [1]. Thus, we compare the frequency with which an action's key appears in the ciphertext to the predetermined frequency that we have for the action's value. The closer the two frequencies are, the more likely it is that the mapping is correct, so we assign lower costs to frequencies that are close together and higher costs to frequencies that are far apart. We tried three different approaches for assigning costs. In the first approach, we merely took the absolute value of the difference between the two frequencies. In the second approach, we took the quotient of the two frequencies (larger frequency divided by smaller frequency). In the third approach, we tried using the error margin, using the ciphertext frequencies as the observed frequencies and the predetermined frequencies as the expected frequencies.

2.2 Example

For an example, let's say we have the ciphertext "JNNJO". Since the alphabet has 26 states and quite a few states are generated by A* search, we'll simplify this example to only consider the possibilities 'E', 'T', and 'H'. We would start with the empty state, {}, and add it to a queue with a cost of 0. We then remove it from the front of the queue and look at the first unassigned letter, 'J'. Since 'J' is the most frequent letter in the ciphertext, showing up 40% of the time, mappings from 'J' to more frequent letters have lower costs. Thus we add {'J' : 'E'} to our queue with cost $40/12.2 = 3.3$, and proceed to add the ({'J' : 'T'}, 4.4), and ({'J' : 'H'}, 6.8). Next, we remove ({'J' : 'E'}, 3.3) from the queue, and try to assign a letter to 'N'. There are only 2 possible actions now – {'N' : 'T'} and {'N' : 'H'}. We note that the latter is unlikely because it creates the bigram 'HH', which is very uncommon. Thus, we might enqueue ({'J' : 'E', 'N' : 'T'}, 7.7) and ({'J' : 'E', 'N' : 'H'}, 15). We continue in this manner, and result in the following steps:

N	Dequeue	Enqueue
1	({}, 0)	({'J' : 'E'}, 3.3) ({'J' : 'T'}, 4.4) ({'J' : 'H'}, 6.8)
2	({'J' : 'E'}, 3.3)	({'J' : 'E', 'N' : 'T'}, 7.7) ({'J' : 'E', 'N' : 'H'}, 20)
3	({'J' : 'T'}, 4.4)	({'J' : 'T', 'N' : 'E'}, 7.7) ({'J' : 'T', 'N' : 'H'}, 25)
4	({'J' : 'H'}, 6.8)	({'J' : 'H', 'N' : 'E'}, 10) ({'J' : 'H', 'N' : 'T'}, 15)
5	({'J' : 'E', 'N' : 'T'}, 7.7)	({'J' : 'E', 'N' : 'T', 'O' : 'H'}, 20)
6	({'J' : 'T', 'N' : 'E'}, 7.7)	({'J' : 'T', 'N' : 'E', 'O' : 'H'}, 12)

Finally, ({'J' : 'T', 'N' : 'E', 'O' : 'H'}, 12) is dequeued, and we are done. This state has the algorithm's guess for the reverse mapping from the substitution alphabet to the regular alphabet, which decodes the ciphertext "JNNJO" to "TEETH".

III Preliminary Results

Our preliminary implementation has a significant problem with runtime. For some reason, using the quotient approach to calculating unigram costs takes a long time. For the purpose of getting results, we halted A* search by modifying the goal to require a lower number of assignments. When using quotient for cost and stopping after 5 assignments, our algorithm took 14 seconds and got 2 assignments correct. When using absolute difference and 12 assignments, our algorithm took 15 seconds and got 2 assignments correct. When using error margins and 12 assignments, our algorithm took 30 seconds and got 2 assignments correct.

IV Future Work

Our results indicate that our algorithm still leaves a lot to be desired. There are still many things we can do to fine tune our assessment of how much an action should cost. First, we may still be able to improve on how we calculate unigram costs. Other obvious considerations are bigram frequencies and perhaps even frequencies for longer n-grams. Whether or not a word in the ciphertext has a chance of being an English word is another important consideration, because in our previous example, 'TEET_' and 'ETTE_' had equal weights, when it is clear that 'TEET_' is more likely. The location of an n-gram in a word is also important. For example, while 'THE' is a common trigram, in a 4-letter word, it is much more likely to appear at the beginning than in the end – compared to 'THE_', there are not a lot of ways to complete '_THE'.

Related to the location of an n-gram in a word is the fact that 1-, 2-, and even 3-letter words have significantly different frequencies than their general unigram, bigram, and trigram frequencies. 'TH' is about 3 times as common as 'TO' [1], but 'TO' is a word and 'TH' is not. Starting with decisions that take these words into considerations optimizes our algorithm by heavily penalizing mappings that are highly unlikely, which more closely matches what an expert human cryptographer would do. This is important because our algorithm currently takes a long time to run. If we were to do something similar to beam search and limit the length of our queue by discarding the most costly assignments, we could significantly improve the runtime.

Due to the low accuracy and long runtime of our current approach, we may consider other algorithms. We may also stick to regular substitution ciphertexts and forego noise, as noise would greatly complicate what seems to be already a complex problem.

V References

[1] Substitution Ciphers, Cornell Math Department
http://www.math.cornell.edu/~mec/2003-2004/cryptography/subs/substitution.html