Jadon Schuler
CSCE 557
Sep 10, 2020
Program 1

Substitution Cipher Project

Because I have provided two different programs, I will begin by describing the simpler of the two, the mostly manual solver. This program starts by counting the frequency of the letters in the cipher text, then printing this for the user. From here, the user can make educated guesses about which letters match to slowly generate the plaintext. Using the frequency of letters in the English language, it should be possible to determine which letter matches with 'e.' From there, 't' and 'h' should also be the next easiest letters to solve, and from there, trial and error can be used. The program also allows for the undoing of a match in case the user makes a mistake. This is the program I used to decode the cipher text for the first time, thereby allowing me to determine which words were in the dictionary and which needed to be ignored by my automated solver (more on this later).

Next, the automated solver. This program begins much the same way, counting and displaying the letter frequency of the cipher text. It then prompts the user for any cribbing input they would like. From here, the program diverges. First, we create a dictionary where the keys are each unique character in the cipher text, and the values are strings of the possible matching plaintext characters. We then sort the letters in decreasing frequency order and solve them in that order. Now we are ready to decrypt the text.

We begin by taking a character from the frequency-sorted list and adding it to the list of characters to test. Then, we test each possible match for each cipher character in this list individually. In order to do this, we loop through the cipher words, and add any that are complete (complete being defined as every character in a "complete" word appears in the list of characters to test) to a separate list, as long as a copy of that word does not already exist in the list. We then sort this new list of complete words by length so that we test short words first to maximize efficiency.

For each complete word, we then recursively generate all possible permutations of the plaintext for that individual word, being sure to keep the letter we are testing constant. In order to maximize efficiency, once a valid word is found we immediately skip to the next word. Similarly, if a valid word cannot be generated from *any* of the permutations, we immediately

break the loop and eliminate that plaintext character from the list of possible matches of the cipher character we are testing.

In this way, as more and more cipher characters are added to the list being tested, more and more words become complete and the loop continues to pare down the possibilities until the plaintext is generated.

However, there are two things to note about this automated solver. First, because of its zero-tolerance policy when it comes to errors, if *any* of the enciphered words does not appear in the dictionary, the program may not complete as intended. As a workaround I added a feature where if a word in the cipher text is prepended with a star (‘*’), the program ignores that word entirely when checking for valid words. For this reason, after decoding the text using my first program, I did just that to a second cipher text file I modified myself in order to test my automated solver. The second is that since the order of letters is decided based purely on frequency, a naïve order may be chosen. If this happens, while the program will eventually decode the message, it may not run in a reasonable amount of time for other cipher text files. In terms of the file I was assigned, the program should complete within 3 to 5 minutes.