CS4000

Homework # 5: Unbreakable Crypto and MPI

due Thursday, March 24th, 2016, 11:59 p.m.

(50 pts.)

Introduction

It is possible to encode secret messages in a way that is virtually unbreakable. The following approach is well-known. Construct a random key of bits X that is as long as the secret message Y (of bits) that you wish to send to your friend. For each bit of Y, you compute the exclusive OR of that bit of Y with the same bit of X, and you send this encoded message Z to your friend. From an information theoretic point of view, this type of encrypted message, called a one-time pad (see, e.g., the wiki page on one-time pad) is unbreakable. Indeed, this type of cryptography has been used in the past to transmit secret messages. Now, as long as your friend knows the random key X, your friend can take the exclusive OR of X and Z to construct the secret message Y. Hence, the encryption and decryption algorithms are *exactly* the same.

Now, it is hard to send big secret keys to one another. So, you decide to share a small secret key with your friend. For example, you tell your friend the secret key "dinosaur," and then encrypt your message by applying the C++ exclusive or operator (`^`) to each character in your message. Since your message is longer than the word "dinosaur", you encrypt the ith character of the message using the `i mod 8`th character of the secret key. Your encryption/decryption algorithm is pretty simple. Attached to this homework is an implementation of this algorithm in the file `encrypt.cc`. As a test, you encrypt the file `message1.txt` to create the file `message1.enc`. Running the same program on `message1.enc`, using the same secret key "dinosaur" gives you the original message back.

Breaking the code

Now, I used your same program to encrypt a secret message. My secret message is stored in the file `message2.enc`. The file is the *ciphertext*. Your job is to find the *plaintext*. You know three things that will help you find the *plaintext*.

1. The ciphertext was created by the program `encrypt.cc`.
2. The secret key is a single dictionary word from `words`.
3. The plaintext is English text.

Given these three facts, you can crack the code by decrypting the message using each of the 118K possible dictionary words (in the file `words`). One of these 118K decrypted messages is the correct one. But, which one is it? Since you know that the plaintext is written in English, the frequency of characters in the decrypted message must be similar to the standard frequency of characters in English text. So, your "cracking" code will output to find the decrypted message that minimizes

$$\sum_{i=0}^{255}(plaintextfreq[i] - standardfreq[i])^2. \quad (1)$$

For this project, we will use the frequencies contained in the file `standard_freq.dat`. (These frequencies were computed by analyzing several plays by Shakespeare.)

1. (10 pts.) Write a program that reads in the standard frequencies of characters in English (`standard_freq.dat`), the ciphertext (`message2.enc`), the list of dictionary words (`words`) and outputs the plaintext (decoding using a key from the dictionary) that minimizes equation (1).

2. (10 pts.) Decode the message `message2.enc` into the correct plaintext using your code from 1. (Hint: the plaintext is relatively well-known. Your code should be able to do this in less than 30 seconds.)

3. (5 pts.) What was the password?

A New Code and MPI

Now, you want to make your messages more secure. So, you decide to use two words from the dictionary (concatenated together) as your secret key.

3. (25 pts.) Write an MPI program that finds the plaintext/password pair that minimizes equation (1) when the password is assumed to be a a pair of dictionary words from a dictionary (`small_words`) that contains around 10,000 words. The input to your MPI program are the file names that store the (1) character frequencies database, (2) the ciphertext, and (3) dictionary.

4. (5 pts. extra credit) Output the password/ plaintext pair that minimizes equation (1) for the file `message3.enc` and the dictionary `small_words`. The sequential version of this progam should take around 310 minutes (over 5 hours). A parallel version running on all of the bin and sp-machines might get done in < 5 minutes.

Hint: Make sure to run your program on easier test cases before trying to decode `message3.enc`.