Breaking a Simple Substitution Cipher

Assignment 4 CS472 F20 DUE: Sun Oct 18, 2020 at 5pm PT

REWARD: 200 points

1 The Problem

In this assignment we will try to break simple substitution ciphers and learn the basic format of a permutation genetic algorithm. Allow plenty of time for this assignment.

You will write a program called `decipher` that will take a coded message in on stdin. It will break it with the help of the English Digraph Frequency Table found in the file `englishFreq.cpp`. The command takes no argument during the testing.

Try to match the output format and quality of answer in the results. You don't have to get an exact agreement on the keys. For larger texts such as over 1000-2000 characters, I would expect to get the same answer. For shorter texts a few characters different in the keys.

The Problem

The encoded message that is sent is called the **cipher text**. The UNencoded message is called the **clear text**. The goal will be to write a GA to crack simple substitution ciphers. Given the cipher text it will return the key and the clear text.

A simple substitution cipher is one in which each possible letter in the clear text has a unique translation into one character in cipher text (1-1 onto mapping). To encrypt the message one simply makes a one for one substitution of the letters in the clear text with the ones in the cipher text using a key.

$$clear\ text \xrightarrow{key} cipher\ text$$

The key is the mapping expressed as the alphabet of the clear text translated into cipher text alphabet. In our case this is the encoding for the 26 lowercase letters: "abc...z". IMPORTANT: we want the key that would be used to translate the clear text to the cipher not the inverse which is the key to translate into the clear. Make sure everywhere you use the encryption key and not the decryption key! Printing the decryption key will NOT match.

For example: The clear text:

```
``A dog looks up to a man. A cat looks down on a man. But a patient
horse looks a man in the eye and sees him as an equal.''
```

The key is the second line below

```
abcdefghijklmnopqrstuvwxyz
cpmtzkrhlsquniebdfaygwovjx     <---   the encryption key
```

The final cipher text is then broken up in the 5 letter blocks:

```
cteru eeqag byecn cicmc yueeq ateoi eicnc ipgyc bcylz iyhef
azuee qacnc iliyh zzjzc itazz ahlnc acizd gcu
```

In this example, we encoded the message as follows. The clear text has had all uppercase letters changed to lowercase letters and all whitespace and punctuation removed. It is then enciphered with a substitution cipher determined by the key. Then the message is broken up in the traditional way of blocks of 5 letters each to hide the word breaks.

You will be given several enciphered texts each enciphered with a potentially different key. Your program must determine what the key and clear text is by evolving a key that optimizes a fitness function. Longer texts will be easier to solve than shorter ones. (Why?) Your algorithm should be able to crack the codes completely for longer texts and only be off by a few infrequently used letters for shorter texts. You are not expected to get the perfect key in all cases. (Why?) You are not to attempt to recover the word breaks in a message, only the key.

Use the fitness function described below and the digraph frequency table supplied in the resources on the class page to formulate a fitness. You should write a GA of your choice (steady state or generational) and a locus based permutation xover and mutation of your choice. You may optionally perform a simple hill climbing local search if you think that will improve things. This can be done periodically or at the end as you choose. Experiment! The goal here is not to solve these perfectly, but to see how well they can be solved using these algorithms and what are the gotchas. Again, you should come within a few letters or perfect for the key and those characters that are different should be infrequent letters like j, v, k. The longest messages you should miss by only a couple of infrequent letters worst case.

Feel free to augment the fitness function with a punishment function for bad choices as described in class and see if you can get improved performance in speed or quality. Be prepared to talk about your results in class.

The Fitness Function

To compute the fitness function for a key (a permutation of 26 letters) a **contact table** or **digraph frequency table** for English is used. A frequency table is a 26×26 matrix F where each element F_{ij} is the number of times each possible ordered pair of letters ij occurs in a sample of text i.e it is a count of occurrences of pairs of letters. A frequency table for English is supplied in `englishFreq.cpp`. Read the comments in the file. It is derived from a large sample of English text called a corpus.

Convert the table to a contact table which contains the probability of the pair occurring in English that is E is the normalized version of F.

$$E_{i,j} = \frac{F_{i,j}}{\sum_{a,b} F_{a,b}} \qquad \forall i, j$$

The fitness of key K is derived by comparing the contact table for a large sample of English with the contact table from a message translated with the key K. Here is how the comparison is done: Let E be the contact table for English and C the contact table for cipher decoded by key K. VERY IMPORTANT: that you only need to compute C once for each cipher you are solving!!!

Let $e(i)$ be a function that encodes the letter i as defined by the key K. For the example key above $e(a) = c$ and $e(b) = p$.

Once the cipher text is read in you create the contact table for the cipher text. Since the counts in the contact table are just counts it must be normalized to compare with the contact table E. **They both need to be normalized**. Do this only once or your program will be too slow to finish. The $fit(K)$ is the fitness of the encoding key K. It is the negative of the square of the Euclidean distance. It is computed as the sum over all ordered character pairs ij as follows:

$$fit(K) = -\sum_i \sum_j \left(E_{i,j} - C_{e(i),e(j)}\right)^2$$

where $e(i)$ and $e(j)$ are the encoding of the characters using the key K. E is the normalized English digraph table and C is the normalized digraph table from the encrypted text. NOTE: precompute what you can so you don't do hundreds of divides for ever fitness eval. IMPORTANT: The smaller the fitness the better the match so to make it a maximization problem we take the negative of the Euclidean distance. As the key changes, the encoding changes and so does the fitness of the key.

An alternate and fairly successful fitness function is the Bhatthacaryya distance and is computed as:

$$fit(K) = \sum_i \sum_j \sqrt{E_{i,j} C_{e(i),e(j)}}$$

which must be maximized as is.

Consider what other rewards and punishments you can put into the fitness function. Discuss the plain fitness presented above and any others you might be interested in.

What to Turn In

You should turn in your code in a tar file along with a report. Your code and makefile should create a program named `decipher`. I will grade your program by compiling it and running it against test files used as standard input. For example:

```
decipher < x4115.txt
```

The exact format for your output of the key is: `** yourSubmitName key` For example for the key printed out for the above for user meerkat is:

```
** meerkat cpmtzkrhlsquniebdfaygwovjx
```

This should be followed by a decoding of the message using the key.

On the lines that follow you can output whatever else you want as long as it does not begin with **. My code will be looking for the ** marker. There will be a time limit on execution. Be sure you write your fitness function efficiently!

Note that since there is no understanding of English by your program you should be able to send it a long plain English text and it should find the key is `abcdefghijklmnopqrstuvwxyz`. This is a quick and easy test that your program is working. It is fun to watch the key converge to "a through z" if you print out the best key every generation.

2 Submission

Homework will be submitted as an uncompressed tar file to the homework submission page linked from the main class page. You can submit as many times as you like. The LAST file you submit BEFORE the deadline will be the one graded. For all submissions you will receive email giving you some automated feedback on the unpacking and compiling and running of code and possibly some other things that can be autotested. I will read the results of the runs and the reports you submit.

Have fun.