CRYPTANALYSIS OF A CLASS OF CIPHERS BASED ON FREQUENCY ANALYSIS

1 Introduction

1.1 Team Members

Sandeep – srm9547
Ryan – rsk457
Thangam – ta2313

Ryan worked on Approach1. Thangam worked on FreqDict class and wrote the report. Sandeep worked on Approach2.

1.2 Approach - 1 - Ryan

Frequency analysis takes advantage of the fact that some characters appear more than others.

Our Approach:

In our approach, the program first returns an initial guess by the means of frequency analysis. The returned guess is then compared to each and every plaintext candidate by iterating over the dictionary. While this comparison is carried out, any characters in the guess that do not match the positions in the plaintext dictionary are excluded

This value that we obtained by excluding the characters that don't match can be used as a parameter that will help us determine whether the plaintext is accurate or not. The plaintext guess that has the highest score (low parameter value) will be assumed to be correct with a high probability.

2. Method

```
map<char,
char>
keygen(string
cipher,
string
candidate) {


                        map<char, char> out;

                        auto freqtable = freqsort(candidate);
                        auto sorted = freqsort(cipher);

                        for(int i = 0; i < sorted.size(); i++) {
                                out[sorted[i].first] =
                freqtable[i].first;
                        }

                        return out;

                }
```

The keygen function above, first sorts the candidate plaintexts and the ciphertexts by their frequencies as the frequency of the plaintext characters may be permuted. Then it returns a key based on the frequency.

```
int
noiseamount(string
```

```
cipher, string
candidate,
map<char,char>
key) {

                        string d_cipher = decrypt (cipher, key);

                        int i_cipher = 0;
                        int i_candidate = 0;
                        int noise = 0;

                        while (i_cipher < d_cipher.size()) {
                               if(d_cipher[i_cipher] == candidate[i_candidate]) {
                                      i_cipher ++;
                                      i_candidate ++;
                               } else {
                                      i_cipher ++;
                                      noise ++;
                               }
                        }

                        return noise;

                }
```

The function in the code above is what we used to calculate the noise amount (the parameter or score that is used in deciding which is the most accurate plaintext). This relies on the plain text dictionary.

```
if(d_cipher[i_cipher] == candidate[i_candidate]) {
       i_cipher ++;
       i_candidate ++;
```

The snippet of the code above compares individual characters in the cipher and plaintexts. If equal, it doesn't increment the noise parameter, if not, it increments the noise parameter.

```
string
analyze(strin
g cipher) {


                        string guess;


                        ifstream infile("dictionary_1.txt");

                        int bestnoise = INT_MAX;
                        int c_noise;

                        string candidate;

                        while(getline(infile, candidate)) {
                                if(candidate[0] <= 'z' && candidate[0] >= 'a') {
                                        auto keyout = keygen(cipher, candidate);
                                        c_noise = noiseamount(cipher, candidate,
                keyout);
                                        if(c_noise < bestnoise) {
                                                bestnoise = c_noise;
                                                guess = candidate;
                                        }
                                }
                        }

                        return guess;

                }
```

The function above compiles all the functions used till now to make a guess of the plaintext. It returns the guess with the lowest noise or the highest score as the probable plaintext