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.

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.

2.1 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(string 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

3 Approach - 2 - Sandeep

To get a better frequency analysis, we created a class FreqDict that builds a map of single letters as well as digraphs.

We can get an estimate of how many random characters the input has by subtracting its length from the texts in the dictionary.

We use this estimate to apply different strategies, for low random characters (about 10% of the length of the text) we can simply use the sum of the difference of the single letter frequencies of the cipher and the dictionary texts and choose the candidate with the lowest difference.

For higher randomness, we assume that the distance between the characters' appearance isn't gonna change by a big factor. We calculate the sum of the distances of the 3 most common frequent characters for both the cipher and the dictionary texts. We subtract these sums to estimate the relative distance factor.

We output our guess as the ciphertext with the least relative distance.

We utilize both these strategies based on the number of random characters.

3.1 Method

In the code snippet, we are using a string and it's frequency map to get the relative distances of the character with a frequency sorted order of freqIdx.

The higher the value of freqIdx, the lower the frequency of that character.

Similarly we calculate the relative distances of diagraphs.

```
vector<int> cal_reld_single(string text, FreqMap &freqMapObj, int freqIdx) {
 vector<int> reld_dists;
 vector<pair<char, int>> freqTable = freqMapObj.getSingFreqSort();
 int lastIdx = -1;
 for (int i = 0; i < text.length(); i++) {
   if (text[i] == freqTable[freqIdx].first) {
     reld_dists.push_back(i - lastIdx);
     lastIdx = i;
   }
 }
```

```
 return reld_dists;

}
```

We then calculate the difference between these relative distances.

```
int get_reld_diff(vector<int> reldDistCipher, vector<int> reldDistCandidate) {

 int i = reldDistCipher.size(), j = reldDistCandidate.size();

 int diff = 0;

 for (int k = 0; k < min(i, j); k++) {

   diff += reldDistCipher[k] - reldDistCandidate[k];

 }

 if (i < j) {

   for (int k = i; k < j; k++) {

     diff += reldDistCandidate[k];

   }

 } else {

   for (int k = j; k < i; k++) {

     diff += reldDistCipher[k];

   }

 }

 return diff;

}
```

And finally iterate over all the plaintexts to calculate their reld values and return the one with the least reld value.

```


string analyzeReld(string cipher, vector<string> &candidates) {


```

```
FreqMap cipherMap(cipher);



int best_reld = INT_MAX;

string best_candidate;

for (int i = 0; i < candidates.size(); i++) {

  FreqMap candidateMap(candidates[i]);

  int reld_diff_avg = 0;

  for (int j = 0; j < 3; j++) {

    vector<int> reld_dists = cal_reld_single(cipher, cipherMap, j);

    vector<int> reld_dists_candidate =

        cal_reld_single(candidates[i], candidateMap, j);

    reld_diff_avg += get_reld_diff(reld_dists, reld_dists_candidate);

  }

  for (int j = 0; j < 2; j++) {

    vector<int> reld_dists = cal_reld_double(cipher, cipherMap, j);

    vector<int> reld_dists_candidate =

        cal_reld_double(candidates[i], candidateMap, j);

    reld_diff_avg += get_reld_diff(reld_dists, reld_dists_candidate);

  }



  if (abs(reld_diff_avg) < best_reld) {

    best_reld = abs(reld_diff_avg);

    best_candidate = candidates[i];

  }

}



return best_candidate;
```

```
}
```

We use these two strategies together with the following algorithm

```
if ((float)(cipher.length() - candidates[0].length()) /

        candidates[0].length() <

    0.15) {

  return analyzeNoise(cipher, candidates);

}

return analyzeReld(cipher, candidates);
```