Substitution Cipher

Kevin Lin, Kong Huang

March 15, 2015

Contents

Chapter 1

Programming Report

1.1 Team Members and Specification Changes

Team Members:

Kevin Lin

- Created the project program and handled the coding aspects
- Wrote sections 1.1 and half of 2.1 of the report

Kong Huang

- Helped with algorithms and decryption schemes through pair programming
- Wrote sections 1.2 and half of 2.1 of the report
- Formatted the report in LaTeX

No modifications were made with respect to the following specifications outlined in the project:

Your program should take as input from stdin:

1. the number t of key symbols,
2. an L-symbol challenge ciphertext,

where each symbol is either a space or one of the 26 lower-case letters from the English alphabet (thus, not a special character, punctuation symbol or upper-case letter). Your program should return as output a guess for which L-symbol plaintext was encrypted, where again each symbol is either a space or one of the 26 lower-case letters from the English alphabet (thus, not a special character, punctuation symbol or upper-case letter). A text file Dictionary1 containing a number u of L-symbol candidate plaintexts will be provided to you, and you should feel free to use its content as part of your code. A text file Dictionary2 containing a number v of English words will be provided to you, and you should feel free to use its content as part of your code.

Your executable file should be named "<last name1>-<last name2>-decrypt". Upon execution, it should obtain the above inputs 1,2 from stdin, and finally return the output plaintext on stdout within x minutes (or else it will be declared to default to an incorrect guess); most likely, we will choose x = 2.

Your program will be run using different parameters (L=100, u=200, v=100, and t between 1 and 24), and on a number of challenge ciphertexts, each computed using a potentially different encryption scheme. Each ciphertext will be computed from a plaintext selected in one of the following two ways:

1. randomly and independently choosing one of the L-symbol plaintexts in Dictionary1 or
2. concatenating words randomly and independently chosen from Dictionary2 (any two words being separated by a space, until one has an L-symbol plaintext).

1.2 Cryptanalysis Approach Used in the Program

What we know:

- The message space and ciphertext space are the set $\{< space >, a, .., z\}^L$. In other words the message m can be written as m[1],...,m[L], where each m[i] is in {(space>,a,..,z}, and the ciphertext c can be written as c[1],...,c[L], where each c[i] is in {<space>,a,..,z}
- The key space is the set $\{0, .., 26\}^t$. In other words the key k can be written as k[1],...,k[t], where each k[j] is in {0,..,26}, for j=1,..,t.
- The encryption algorithm computes each c[i] as equal to the (lexicographic) shift of m[i] by k[j(i)] positions, where the computation of each j(i) is left unspecified and may depend on i,t,L. In other words, each ciphertext symbol c[i] is the shift of the plaintext symbol m[i] by a number of position equal to one of the key symbols, which symbol being chosen according to an undisclosed, deterministic, and not key-based, scheduling algorithm that is a function of i, t and L.

Strategy for Dictionary 1:

Since each letter can be mapped to a corresponding integer value, you can compare the given ciphertext with a possible plaintext using their numerical values in order to analyze the different shifts. The letter to value conversion goes as follows:

A	B	C	D	E	F	...	X	Y	Z	' '
0	1	2	3	4	5	...	23	24	25	26

For Dictionary 1, the plaintext is the entire line. Therefore, all that is needed to be done is to do a direct comparison between the plaintext line and the ciphertext line. Given the strings of the two lines, our strategy was to compare

each corresponding letter. In other words, dict1[0] will be compared with cipherText, dict1[1] will be compared with cipherText, etc. The comparison is done by subtracting the numerical value of the plaintext with the ciphertext. There is an error check where if the subtraction results in a negative number, 27 is added (the size of the table). The resulting value is inserted into a temporary set, in order to ensure no duplicates. If the size of that set (aka the number of unique characters) is less than or equal to the provided t, you have found the plaintext that corresponds to the ciphertext.

The code that handles this comparison can be seen here:

```
bool test1helper(const string &a, const string &b, int t){
   set<int> temp;
   int val1 = 0;
   int val2 = 0;
   for (size_t i = 0; i < a.length(); i++){
      val1 = letter_to_value(a[i]);
      val2 = letter_to_value(b[i]);
      if (val1 - val2 < 0){
         temp.insert(val1 - val2 + ALPHABET);
      }
      else {
         temp.insert(val1 - val2);
      }
   }
   return (temp.size() <= t);
}
```

Strategy for Dictionary 2:

The same strategy used for Dictionary 1 can also be applied to Dictionary 2. Therefore, much of the comparison code will remain the same. In fact, the comparison code looks like this:

```
bool test2helper(set<int> &duplicates, const string &a, const ↩
    string &b, int t, int pos){
   int val1 = 0;
   int val2 = 0;
   for (size_t i = pos; i < a.length(); i++){
      val1 = letter_to_value(a[i]);
      val2 = letter_to_value(b[i]);
      if (val1 - val2 < 0){
         duplicates.insert(val1 - val2 + ALPHABET);
      }
      else {
         duplicates.insert(val1 - val2);
      }
   }
   return (duplicates.size() <= t);
}
```

However, as you can see, the biggest difference is that we are also passing in where we are in the dictionary, as well as the set of duplicates. This is because we have to concatenate the words so it would compare the ciphertext

with them one by one. As we compare the words with the ciphertext, if there is a reasonable assumption that the word is the correct one, we append it to a custom string we defined as 'guess.' The decryption algorithm recursively calls itself until the decrypted plaintext is found. The algorithm runs depth-first rather than breadth first, and the code can be seen here:

```
bool test2(set<int> duplicates, const string &guess, string dict2[], int dict2Length, const string &cipherText, int t){
    if (guess.length() == L){
        cout << "The decrypted plaintext is: " << endl << guess << endl;
        return true;
    }
    else {
        int length = guess.length();
        set<int> duplicatesCopy;
        string guessCopy;
        for (size_t i = 0; i < dict2Length; i++){
            duplicatesCopy = duplicates;
            guessCopy = guess;

            guessCopy.append(dict2[i]);
            guessCopy.append(" ");
            if (guessCopy.length() > L){
                guessCopy.resize(L);
            }

            if (test2helper(duplicatesCopy, guessCopy, cipherText, t, length)){
                if (test2(duplicatesCopy, guessCopy, dict2, dict2Length, cipherText, t)){
                    return true;
                }
            }

        }
    }
    return false;
}

// This function examines the ciphertext using the second dictionary
bool test2(string dict2[], int dict2Length, const string &cipherText, int t){
    set<int> duplicates;
    string guess = "";
    return test2(duplicates, guess, dict2, dict2Length, cipherText, t);
}
```

In terms of efficiency, Dictionary 1's decryption algorithm appears to be of $O(n)$ time where n is the total number of sentences in the dictionary. In the case of this project, $n = 200$, so Dictionary 1's decryption algorithm will be extremely efficient. Whereas Dictionary 2's decryption algorithm appears to be of $O(n^m)$ time where n is the total number of words in the dictionary and m is limited by the ciphertext length and the length of the words in the dictionary. In the worst case scenario, m will be half of the ciphertext length (there are

words in the dictionary with only one character). However, for this project, the ciphertext is 100 characters, the shortest word in the dictionary contains 6 characters, and there are 100 words in the dictionary, so $m = \left\lfloor \frac{100}{6+1} \right\rfloor = 14$ and $n = 100$. This is less efficient than Dictionary 1's decryption algorithm and can potentially be very slow given greater m.

Chapter 2

Extra Credit

2.1 Survey on Substitution Ciphers

Substition ciphers are one of the oldest ways to encrypt a message, their main attractions being that they are simple to use and intuitive. There are many variants of the substition cipher, such as monoalphabetic substitutions which include the simple substitution and homophonic substitution ciphers. There are also polyalphabetic substititution ciphers, such as the Vigenére cipher and Enigma (used in World War II). In this survey, we primarily examine monoalphabetic substitution ciphers.

The simple substitution cipher basically consists of substituting every plaintext character for a different cipher text character. While ciphers such as the Caesar cipher shifts each letter in the plaintext a certain number of places down the alphabet, the simple substitution cipher is completely jumbled. However, it offers almost no communication security, and is vulnerable to even the most elementary statistical analysis. An example of the simple substitution key is shown below:

```
plaintext:   ABCDEFGHIJKLMNOPQRSTUVWXYZ
ciphertext:  YBXONGSWKCPZFMTDHRQUJVELIA
```

Using this key, the plaintext HELLO would encrypt to ciphertext WNZZT. Any given plaintext letter always encrypts to the same cipher text letter. This means that the cryptanalyst looking at the word WNZZT will know that the third and fourth letters are the same, even if they do not know what letter it is. With regards to the English language, the simple substitution key space consists of 26! possible permutations of the alphabet. To think about how big this number is, if someone were to check a million keys per second, it would still take over 12 trillion (10^{12}) years to check all possible keys. Therefore, as a brute force attack is clearly infeasible, a good strategy to crack the cipher is through frequency analysis. As observed before, each ciphertext letter has the same frequency in ciphertext as its corresponding plaintext letter has in the plaintext. Once a few of the high-frequency letters have been determined, common letter patterns will become apparent and the key can then be completely recovered.

On the other hand, a homophonic substitution applies a one-to-many mapping as opposed to the one-to-one mapping of a simple substitution. In other words, there may be more than one possible substitution for a given plaintext letter. However, even if there are multiple ciphertexts, each ciphertext symbol can only represent one plaintext letter. An example of the homophonic substitution key is shown below:

```
plaintext:   A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
ciphertext:  D X S F Z E H C V I T P G A Q L K J R U O W M Y B N
                       7       3     9   5 0       4 6
                                     2
                                     1
```

Using this key, any of the symbols P, 9, 2, or 1 can be substituted for a plaintext L, either Z or 7 can be substituted for a plaintext E, and either Q or 0 for a plaintext O. Therefore, the same plaintext HELLO from before can be encrypted as C7P9Q, CZ2P0, or etc. This means that a cryptanalyst looking at this ciphertext will have no indication that the third and fourth characters represent the same plaintext letter. With N distinct ciphertext symbols and using English as the language, the homophonic substitution cipher has the theoretical keyspace of $26^N \approx 2^{5N}$ as opposed to the 26! of the simple substitution cipher. To compare using the person that can check a million keys per second from before, it will take $26^{100}/10^6 = 3.14 * 10^{135}$ seconds (nearly 10^{128} years) to solve a ciphertext with N = 100. As shown, the differences between the keyspaces between the two different ciphers increases exponentially as the number of distinct ciphertext symbols increases. Of course, this means that an exhaustive brute force search is completely out of the question, and cryptanalysis will depend on other methods.

As mentioned, there are many other substitution ciphers available other than these two. There also exists polyalphabetic substitutions where multiple cipher alphabets are used, notably being the Gronsfeld and Beaufort ciphers. Another substitution method is known as polygraphic substitution, where plaintext letters are substituted in larger groups instead of substituting letters individually. While substitution ciphers are no longer in serious use, the concept of substitution ciphers are used in modern cryptography. An example being AES, which can technically be viewed as substitution cipher on an enormously large binary alphabet.

References

[1] Dhavare, Amrapali. "Efficient Attacks on Homophonic Substitution Ciphers." Thesis. San Jose State University, 2011. *SJSU ScholarWorks* (2011): 1-80. *ScholarWorks.* Web. 14 Mar. 2015.

[2] "Homophonic Substitution Cipher." *Practical Cryptography.* Web. 14 Mar. 2015.

[3] Low, Richard M., Amrapali Dhavare, and Mark Stamp. *Efficient Cryptanalysis of Homophonic Substitution Ciphers.* SJSU, n.d. Web. 14 Mar. 2015.

[4] "Simple Substitution Cipher." *Practical Cryptography.* Web. 14 Mar. 2015.

[5] "Substitution Cipher." *Wikipedia.* Wikimedia Foundation, Web. 14 Mar. 2015.