Using Simulated Annealing to Break a Playfair Cipher

Overview

In this lab, we will use the ***simulated annealing algorithm to break a Playfair Cipher***. A common heuristic approach for breaking a cipher is to generate a large number of keys, decrypt a cipher-text with each key and then examine the resultant plain-text. If the text looks similar to English, then the chances are that the key is a good one. The similarity of a given piece of text to English can be computed by breaking the text into fixed-length substrings, called *n*-grams, and then comparing each substring to an existing map of *n*-grams and their frequency. Like all greedy approaches, this process does not guarantee that the outputted answer will be the correct plain-text, but should give a good approximation that may well be the right answer. This technique is very effective for breaking classical ciphers and has been shown to work well for Affine, Autokey, Bifid, Playfair and Four-Square ciphers. The SA algorithm will generate a random key, decrypt and score the resultant plaintext using quadgrams and iteratively make small changes to the key until the annealing cooling schedule reaches zero.

Using *n*-Gram Statistics as a Heuristic Function

An *n*-gram (gram = word or letter) is a substring of a word(s) of length *n* and can be used to measure how similar some decrypted text is to English. For example, the quadgrams (*4*-grams) of the word "HAPPYDAYS" are "HAPP", "APPY", "PPYD", "PYDA", "YDAY" and "DAYS". A fitness measure or heuristic score can be computed from the frequency of occurrence of a *4*-gram, ***q***, as follows: $\mathbf{P}(q) = \text{count}(q) / n$, where *n* is the total number of *4*-grams from a large sample source. An overall probability of the phrase "HAPPYDAYS" can be accomplished by multiplying the probability of each of its *4*-grams:

$$\mathbf{P}(\text{HAPPYDAYS}) = \mathbf{P}(\text{HAPP}) \times \mathbf{P}(\text{APPY}) \times \mathbf{P}(\text{PPYD}) \times \mathbf{P}(\text{PYDA}) \times \mathbf{P}(\text{YDAY})$$

One side effect of multiplying probabilities with very small floating point values is that underflow can occur[1] if the exponent becomes too low to be represented. For example, a Java ***float*** is a 32-bit IEEE 754 type with a 1-bit sign, an 8-bit exponent and a 23-bit mantissa. The 64-bit IEEE 754 ***double*** has a 1-bit sign, a 11-bit exponent and a 52-bit mantissa. A simple way of avoiding this is to get the *log* (usually base 10) of the probability and use the identity *log*(a × b) = *log*(a) + *log*(b). Thus, the score, ***h(n)***, for "HAPPYDAYS" can be computed as a ***log probability***:

$$\log_{10}(\mathbf{P}(\text{HAPPYDAYS})) = \log_{10}(\mathbf{P}(\text{HAPP})) + \log_{10}(\mathbf{P}(\text{APPY})) + \log_{10}(\mathbf{P}(\text{PPYD})) + \log_{10}(\mathbf{P}(\text{PYDA})) + \log_{10}(\mathbf{P}(\text{YDAY})$$

The resource ***quadgrams.txt*** is a text file containing a total of 389,373 *4*-grams, from a maximum possible number of 26^4=456,976. The *4*-grams and the count of their occurrence were computed by sampling a set of text documents containing an aggregate total of 4,224,127,912 *4*-grams. The top 10 *4*-grams and their occurrence is tabulated below:

[1] An excellent analysis of underflow is given by *Goldberg, D., 1991. What every computer scientist should know about floating-point arithmetic. ACM Computing Surveys (CSUR), 23(1), pp.5-48.*

q	Count(*q*)	*4*-gram, *q*	Count(*q*)
TION	13168375	FTHE	8100836
NTHE	11234972	THES	7717675
THER	10218035	WITH	7627991
THAT	8980536	INTH	7261789
OFTH	8132597	ATIO	7104943

The *4*-grams of "HAPPYDAYS", their count, probability and log value are tabulated below.

q	Count(*q*)	Probability(*q*)	$Log_{10}(P(q))$
HAPP	462336	0.000109451	-3.960779349
APPY	116946	0.000027685	-4.557751689
PPYD	4580	0.000001084	-5.964871583
PYDA	1439	0.000000340	-6.467676267
YDAY	108338	0.000025647	-4.590956247
DAYS	635317	0.000150401	-3.822746584

The final score, ***h(n)***, for "HAPPYDAYS" is just the sum of the log probabilities, i.e. -3.960779349 + -4.557751689 + -5.964871583 + -6.467676267 + -4.590956247 + -3.822746584 = **-29.36478172**. A decrypted message with a larger score than this is more "English" than this text and therefore must have been decrypted with a "better" key.

The Simulated Annealing Algorithm

The following pseudocode uses simulated annealing to break a Playfair Cipher, which uses a fixed-length 25 character key.

```
1.  Generate a random 25 letter key called parent.
2.  Decrypt the cipher-text with the key
3.  Score the fitness of the key as logProbability(parent)
4.  For temp = 10  To 0 Step -1
5.    For transitions = 50000  To 0 Step -1
6.      Set child = shuffleKey(parent) //Make a small change to the key
7.      Score the fitness of the key as logProbability(child)
8.      Set delta = logProbability(child) - logProbability(parent)
9.      If delta > 0 Then  //New key better
10.       Set parent = child
11.     Else If delta < 0 Then  //New key worse
12.       Set parent = child with probability e^(-delta/temp)
13.   Next
14. Next
```

Simulated Annealing Algorithm for Keys

The generation of a random 25-letter key on line 1 only requires that we shuffle a 25 letter alphabet. A simple algorithm for achieving this was published in 1938 by Fisher and Yates. The ***Fisher–Yates Shuffle*** generates a random permutation of a finite sequence, i.e. it randomly shuffles an array key of *n* elements (indices 0..*n*-1):

Pseudocode

```
for i from n−1 to 1
  j ← random integer such that 0 ≤ j ≤ i
  swap key[j] and key[i]
```

Java (Version 1)

```
private void shuffle(char[] key){
   int index, temp;
   var random = new Random();
   for (int i = key.length - 1; i > 0; i--){
      index = random.nextInt(i + 1);
      temp = key[index];
      key[index] = key[i];
      key[i] = temp;
   }
}
```

Java (Version 2 Quicker)

```
public void shuffle(char[] key) {
  int index;
  var random = ThreadLocalRandom.current();
  for (int i = key.length - 1; i > 0; i--) {
    index = random.nextInt(i + 1);
    if (index != i) {
      key[index] ^= key[i];
      key[i] ^= key[index];
      key[index] ^= key[i];
    }
  }
}
```

The method shuffleKey() should make the following changes to the key, logically represented as a 5x5 grid, with the frequency given (you can approximate this using Math.random() * 100).

- Swap single letters (90%)
- Swap random rows (2%)
- Swap columns (2%)
- Flip all rows (2%)
- Flip all columns (2%)
- Reverse the whole key (2%)

Node that the semantic network or state space is implicit and that each small change to the 25-character key is logically the same as traversing an edge between two adjacent nodes.

The Playfair Cipher

Polygram substitution is a classical system of encryption in which a group of *n* plaintext letters is replaced as a unit by *n* cipher letters. In the simplest case, where $n = 2$, the system is called *digraphic* and each letter pair is replaced by a cipher digraph. Digraphic ciphers have the advantage of eliminating the possibility of identifying single letters as entities, i.e. confounding monographic frequency analysis. Of course, *digraphic* frequencies can be used, but there are 26x26=676 combinations, making frequency analysis more difficult. Note that the problem state space expands exponentially with *trigraphic* (26^3=17,576), *tetragraphic* (26^4=456,976), *pentagraphic* (26^5=1,1881,376) etc. A 25 unique-letter key has a total of 25! = 1.55×10^{25} permutations. Note however, that the path between a random key and the actual decryption key is only a maximum of $(n * (n + 1))/2 = (25 * (25 + 1))/2 = 325$ moves away.

The key for a Playfair cipher is a sequence of 25 unique letters that forms a 5x5 matrix. *Note that the letter J is not included.* The encryption / decryption process works on diagraphs as follows:

1. Prime the Plaintext

Convert the plaintext into either upper or lower-case and strip out any characters that are not present in the matrix. A regular expression can be used for this purpose as follows:

```
String plainText = input.toUpperCase().replaceAll("[^A-Za-z0-9 ]", "");
```

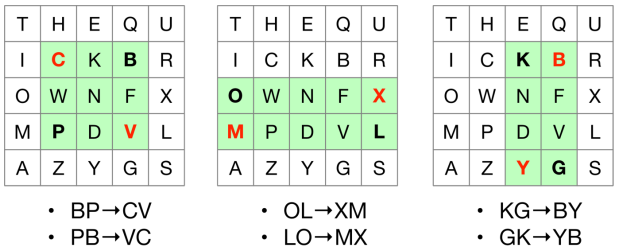
Numbers and punctuation marks can be spelled out if necessary. Parse any double letters, e.g. LE**TT**ERKE**NN**Y and replace the second occurrence with the letter X, i.e. LE**TX**ERKE**NX**Y. If the plaintext has an odd number of characters, append the letter X to make it even.

2. Break the Plaintext in Diagraphs

Decompose the plaintext into a sequence of diagraphs and encrypt each pair of letters. The key used below, THEQUICKBROWNFOXJUMPEDOVERTHELAZYDOGS, is written into the 5x5 matrix as THEQUICKBROWNFXMPDVLAZYGS. The plaintext sequence of characters, "ARTIFICIALINTELLIGENCE", is broken into the diagraph set {AR,TI,FI,CI,AL,IN,TE,LL,IG,EN,CE} and is encrypted into the ciphertext "SIIOOBKCSMKOHQSLBAKDKH" using the following rules (reverse to decrypt):

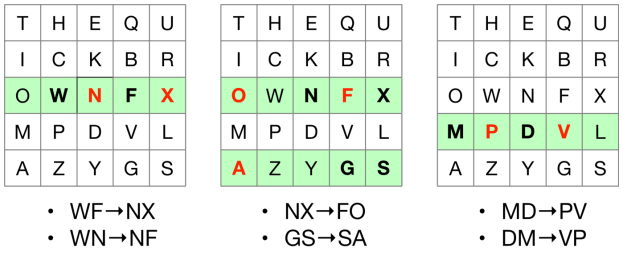
Rule 1: *Diagraph Letters in Different Rows and Columns*

Create a "box" inside the matrix with each diagraph letter as a corner and read off the letter at the opposite corner of the same row, e.g. AR→SI. Reverse the process to decrypt a cypher-text diagraph.



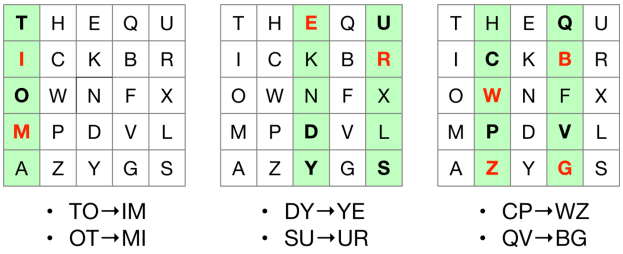
Rule 2: *Diagraph Letters in Same Row*

Replace any letters that appear on the same row with the letters to their immediate right, wrapping around the matrix if necessary. Decrypt by replacing cipher-text letters the with letters on their immediate left.



Rule 3: *Diagraph Letters in Same Column*

Replace any letters that appear on the same column with the letters immediately below, wrapping back around the top of the column if necessary. Decrypt by replacing cipher-text letters the with letters immediately above.



Exercises

- Download the Zip archive of resources for this lab from Moodle and extract the contents to a directory. You should see the following files and folders:
    - **4grams.txt:** a text file containing a total of 389,373 4-grams.
    - **cyphertext.txt:** This document contains the cipher-text of Chapter 1 of The Hobbit, encrypted with the key **THEQUICKBROWNFXMPDVLAZYGS**. The total number of characters in the text is 35,713.
    - **ie:** A directory containing the fully packaged source code of the cypher breaker.

- Compile the source code and execute the programme with the following command:

    java ie.atu.sw.ai.CipherRunner

- Examine the screen output and the contents of the generated file **out.txt**. You should see enough of the document converted to plaintext to enable you to clearly read the chapter.

- Edit the class ***SimulatedAnnealing*** and change the following:

    - The value of **TEMPERATURE** from 30 to a high value, e.g. 300 or 3000. This will transform the search into a random walk as many worse options will be selected due to the high temperature.

    - The value of **TRANSITIONS** from 100000 to 1000. This will reduce the degree to which local optima can be fully explored at lower temperatures.

    - Experiment with the loop control variable(s) ***temp*** and ***trans***. The latter must allow enough iterations at each temperature so that the system stabilises before cooling down.