Using MCMC to Break Stream Cipher

Stochostic Process: Project 2

CHI-NING,CHOU
PROFESSOR RAOUL NORMAND
May 19, 2015

Abstract

While MCMC (Markov Chain Monte Carlo) is a well-known method for generating random samples of complex distribution, it can also be applied in optimization problem of a difficult target function. In this project, I implemented the famous Metropolis-Hasting algorithm with Gibbs-like sampling mechanism to break stream cipher. The best performance in this project is to break 80.1% of a 1024 bytes in stream cipher.

In the first section, I will briefly introduce the problem setting. And discuss the algorithm I implemented in this project in section 2. The last section is going to evaluate the results and make some comparisons between different parameters setting. The codes and other related material can be found in the appendix.

Keywords: MCMC, Stream cipher

1 Problem Setting

1.1 Stream cipher

Stream cipher is a symmetric key cipher based on the operation of bitwise XOR. That is, the encrypt function xor the key stream with the plain text and generate the cipher text. On the other hand, the decrypt function xor the same key with the cipher text and get the plain text. In real life, the implementation is much more complex. However, the basic idea is roughly the same.

The stream cipher I concerned in this project is very straight forward. I used a fixed length key xoring the plain text as encryption, and do the same operation on cipher text as a decryption.

Note that although the stream cipher I used is no longer used in reality, the concepts is almost the same. To break other stream cipher with the same approach only needs some changes in the function settings and parameters. As a result, for convenient and fast implementation, I chose this simplified version of stream cipher as the encryption/decryption mechanism used in this project.

1.2 How to break?

Basically, I used the intuitive *Frequency Analysis* to break the stream cipher. To simplify the algorithm, I only applied the bigram analysis.

In bigram analysis, the program first get the frequency of every possible two consecutive pairs of characters in the text. This can be done by taking the results that other researcher shared online or do the same analysis on the texts that are similar to the plain text you are going to play with.

After having the (relative) frequency of consecutive characters, the program then calculate the distance between the empirical frequency of the text that is decrypted by a guessed key. Then using a target function to generate a target value. The higher the value is, the closer the empirical frequency and the data frequency are.

1.3 Data

In order to have a large data containing lots of texts, I downloaded the famous novel: *The Hunger Games* as the training text to generate digram frequency and used a small part of it as the plain text. There are 523,003 pairs of consecutive characters in *The Hunger Games.*

2 Mechanism

2.1 Working flow

The working flow of the algorithm is as follow.

First, the pre-processing part will calculate the time for certain pattern shown up in the original data text. Then stored the results in the 128*128 score matrix as a reference for target function.

In the main part of the algorithm, there will be three consecutive subtasks with different parameter settings. In each subtasks there are a series of cycle processes which do the same iterations over and over again. In the following sections, I will dig into more about the details of implementation of each steps.

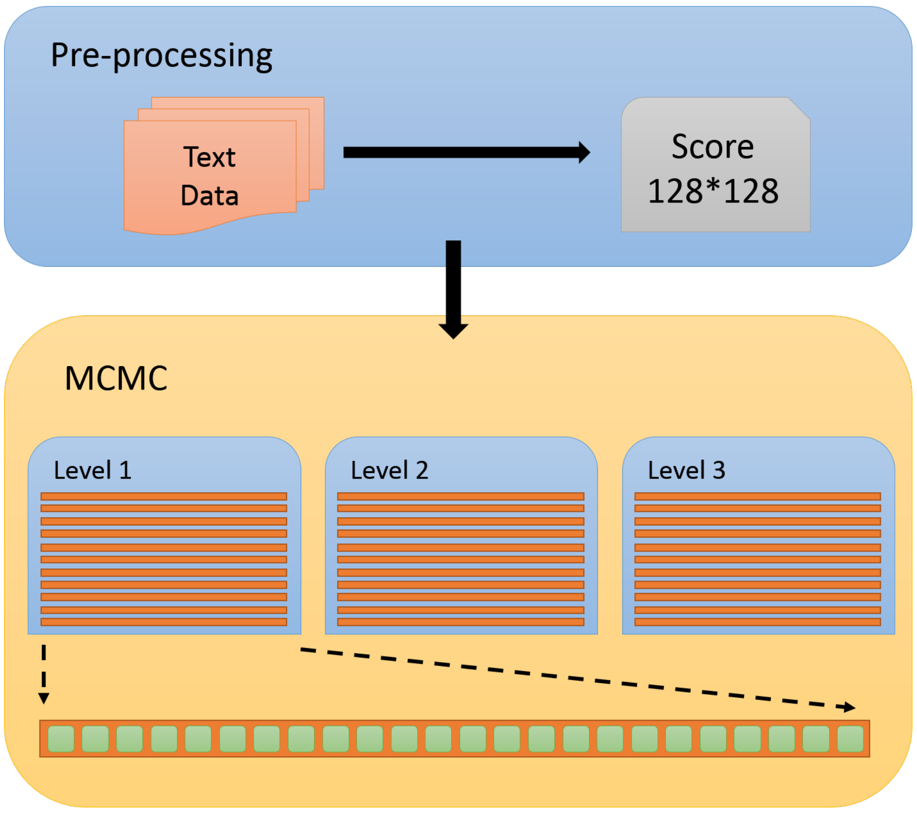


Figure 1: Working flow.

2.2 Pre-processing

In the preprocessing step, a simple program runs through the texts and record the number of appearance of each pair of digram.

2.3 Sub-MCMC

There are three sub-MCMC: level 1, level 2, level 3. Each sub-MCMC contains lots of cycles that keep doing the same operation except that one parameter is different. The parameter: *neighbor* is not the same. *neighbor* indicates the scope of the search region of key. With the observation that after several iterations, the guessed key will be much more close to the real key. It will be much more efficient to search the key in the smaller neighborhood of the current guessed key.

2.4 Cycle

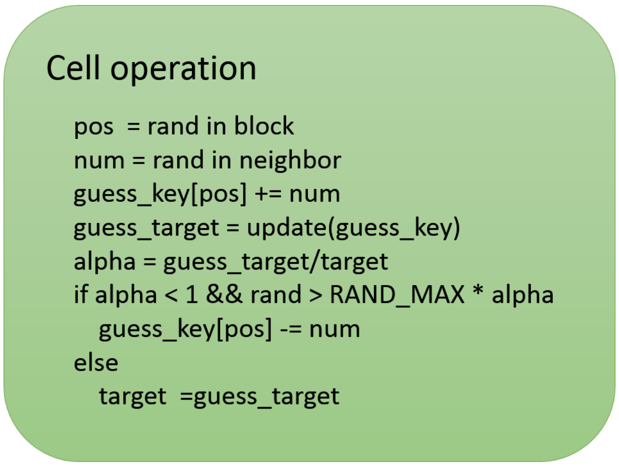
With the idea similar to Gibbs sampling, in each sub-MCMC, the program iterates through the whole key and applies Metropolis-Hasting algorithm only on a small part of the key fragment at one time.

In some sense, a 1024 bytes key, for example, is a variable with 8192 parameters, where each parameter takes values in $\{0, 1\}$. Clearly that it is a gigantic parameter (key) space. As a result, under the observation that the program ran very well on the case with key length 2, I cut the key into pieces of length 2. Then perform cell operations on each of them. Finally, sequentially do the cell operations and make a cycle.

2.5 Cell operation

In each cell, the program runs the Metropolis-Hasting algorithm on the specified key segment. The number of iterations in a call operation is set as a parameter of the algorithm.

The algorithm is as follow.



Cell operation

```
pos  = rand in block
num = rand in neighbor
guess_key[pos] += num
guess_target = update(guess_key)
alpha = guess_target/target
if alpha < 1 && rand > RAND_MAX * alpha
   guess_key[pos] -= num
else
   target  =guess_target
```

Figure 2: Cell operation

2.6 Target function

There are three steps to calculate the target value of a guessed key with respect to a given frequency distribution. First, calculate the digram of decrypted text and normalize both the empirical digram $\hat{D}$ and the given digram D_g. Next, use a distance metric ρ to evaluate the difference between two distribution. Last but not least, Put the result into a non-decreasing kernel ϕ in order to scale the difference.

In this project, I take one norm $\|\cdot\|$ as ρ and $\exp(\text{cot})$ as ϕ. Note that, taking $\exp(\cdot)$ as ϕ makes the ratio between two different distribution larger. Namely, the program will tend to stay at the original state if the difference is not large enough.

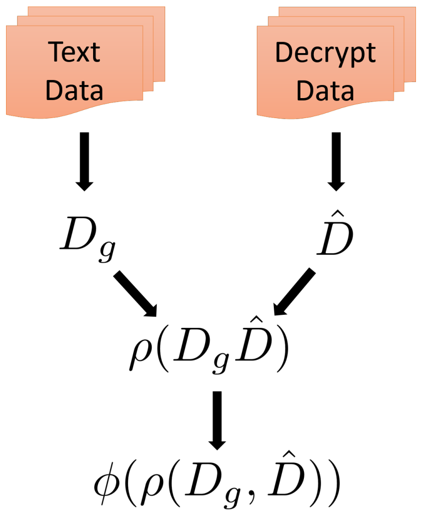


Figure 3: Target function.

3 Evaluation

In the evaluation section, the following will discuss the different effects of some parameters such as *key length*, *text size*, *number of cycles*, *number of iterations*. The criteria for evaluating the performance will be based on the ratio of the decrypted bytes of the original key.

It's quite intuitive that it's more difficult when the length of the key is longer, the size of the text is smaller, the number of cycles or iterations is shorter. Let's see the results and verify the intuition.

3.1 Key length

In this project, the key length I used are: 16, 64, 256, 1024 bytes. The corresponding key spaces are: $3.4 * 10^{38}$, $1.34 * 10^{154}$, $2.8 * 10^{616}$, $1.6 * 10^{742}$. The number of bytes recovered from the program is listed below. Each of the experiment use the largest plain text data and run 100000 iterations with 100 cycles.

	16	64	256	1024
Recovered (%)	87.50%	89.06%	83.98%	68.94%
Avg. error	0.38	0.94	3.59	9.04
Time (sec)	222.8s	234.7s	198.9s	125.6s

3.2 Text size

The total bytes in the original training text: *The Hunger Games* are 523003 bytes. In this project, I use three different size of plain text size to test the program. They are: 2886, 41415, 484886. The key size are fixed to 1024 bytes and the number of iterations is 10000 with cycle 100.

	2886	41415	484886
Recovered (%)	14.64%	57.42%	69.53%
Avg. error	29.36	12.01	7.83
Time (sec)	5.66s	16.74s	121.2s

3.3 Number of iterations

I test the program runs in 20000, 50000, 80000, 100000 iterations. The key size are fixed to 1024 bytes. And I chose the largest text size with cycle 100.

	20000	50000	80000	100000
Recovered (%)	28.41%	46.58%	69.53%	66.40%
Avg. error	17.63	12.34	7.60	7.60
Time (sec)	60.26s	103.37s	121.27s	120.60s

3.4 Cycle

In the following experiment, I use 1024 bytes key, the largest text, 100000 iterations and compares the results with different cycle: 1, 5, 10, 50, 100.

	1	5	10	50	1000
Recovered (%)	30.07%	36.91%	48.14%	70.41%	67.87%
Avg. error	15.34	11.53	9.95	6.65	8.60
Time (sec)	255.32s	245.18s	240.96s	185.89s	126.45s

4 Conclusion

It's not very easy to break a cipher even though it is not very complex like the simple stream cipher in this project. First, one needs to find a good target function to evaluate the guessed key so that the optimization problem can have a better and more meaningful results.

In addition, when using the MCMC method, I found out that the design of algorithm is very important. At the very beginning of this project, I wrote a poor designed program and have a terrible performance, only about 1% of the key can be recovered, which is almost the same as doing nothing. Then I tried to figure out which part can be improved with some observations. And the performance gradually became better and better.

It's a great experience to implement a MCMC method to break stream cipher. However, due to the limit of time, the project is not very complete and rigorous. If I have time one day I have more time, I will try to make it more general and fit the real life problem.

References

1. *Crypto Corner*, http://crypto.interactive-maths.com

2. *The Hunger Games*, Suzanne Collins. 2008. Scholastic. U.S. https://sites.google.com/site/the74thhungergamesbyced/download-the-hunger-games-trilogy-e-book-txt-file

Appendix

The code of this project can be found on Github: https://github.com/jerrychou82/MCMC_Break_Stream_Cipher It's welcome to discuss the code with me!