Weekly reports

Lauri Kangassalo

June 7, 2013

Week 1

During this week I was mostly studying the basic principles of hash cracking and rainbow tables. I have never done anything like this before, but I think I now handle the basic principles of how rainbow tables work.

It is still unclear to me how to optimize my code for cracking hashes, since I don't know how exactly JVM handles memory allocations and such. For example I'm uncertain wether I should use byte arrays in my code, or convert those arrays into long values. I'm sure that as my code progresses I will find an answer to these problems.

I started coding the very heart of the algorithm, the reduction function. It seems to be working correctly, but as stated before, I am uncertain whether it's efficient enough. I also managed to code a class for generating rainbow tables and saving them into a file.

Next I'm going to proceed to the actual hash cracking part.

Week 2

I started this week by coding the class that cracks the hashes *(MD5Crack)*, and got it almost done. However, I discovered that I can't proceed in this part of my program unless I create my own implementation of hashmap/hashset first. Another option would be to create a new class for byte arrays, and I'm worried that it could affect performance, since byte arrays are in such a great role in my software. Besides, I feel that it would be kind of an overkill to create such a class, since all I would need it for would be an equals and a hashcode method.

Since I found myself in a temporary dead-end, for not being able to actually test the password cracking, I decided to code some secondary parts of the program. Firstly, the passwords are no longer fixed in length, for example the user could now choose to include password of lengths 3 to 9 to the rainbow table. This feature is still somewhat buggy, as the program doesn't read the passwords from the rainbow table file correctly. I will get back to this problem as soon as I can get more pressing problems in the code

in order. Secondly, I created a simple user interface for the program, to aid in performance testing. Also, all printing is now handled in classes *UIHelper* and *UI*.

Performance has been tested by hand from the very start of this project. Sadly, I haven't been able to test the hash cracking part, since it doesn't work yet. However, in my experience the table creator *TableCreator* class's performance is sufficient. I have been creating rainbow tables of various sizes, but haven't yet tried to run my program on large enough parameters, since creating a really working rainbow table will take a lot of time and disk space. I will get back to this topic as soon as I can rely on the reading of rainbow table files, so that I don't waste my time creating a table file I can't open in the future.

During this week I also started rereading the materials from Data Structures -course regarding hash tables. I implemented a crude version of a hash table (with insufficient testing), but realized that I cannot make an array large enough / the whole file can't be read to memory at once, since the table files can be tens or even hundreds of gigabytes in size. I have to figure out what I'm going to load to memory and what to keep on the disk, and how to access the data on the disk. I also countered some major performance issues when testing my hash table implementation. This time I used linked lists to deal with collisions, next I'm going to try open addressing - since I won't need to create new object for each value then - and see how it affects performance.

Week 3

This week I mostly concentrated on getting the hash cracking working. I noticed that my code had some major bugs and I've mostly spent my time trying to create rainbow tables with different parameters, and trying to crack some hashes.

First bug I noticed was that reading/writing passwords of different lengths didn't actually work. The endpoint's length didn't match the starting point's, and the passwords were read from the file as if they were all of the same length. This has now been fixed, but there are still issues with the probabilities of successful hash cracking, since the keyspaces of passwords of different length are of different size.

Second bug I found was not actually a bug. I was wondering why my code generates so little different endpoints, and came to the conclusion that the problem must be within the reduction function. Only then it cleared to me that these same endpoints are the collisions mentioned in the papers regarding rainbow table cracking. The answer for this problem was not in the reduction function, but in the parameters of table creation (chains per table and chain length). I found that the optimal values for these parameters

are (roughly):

chains per table 1/4 of the total keyspace size or less

chainlength 1/200 of the number of chains or more

When I'd fixed these bugs, I realized that MD5Crack-class did not compare the hashes correctly. For instance, I didn't extract the original hash's byte array correctly. I learned that hashes are actually string representations of hexadecimal numbers. I also noticed that MD5Crack didn't actually produce the correct chains, but rather just hashed/reduced the endpoint over and over again.

In the end of the day however, I got the cracking to work. It currently cracks passwords of fixed lengths quite proficiently, and cracks about 50% of hashes of passwords with different lengths. I'm not yet quite satisfied with the latter percentage, but I suspect that by adjusting the table creation parameters I can lift the probability to a tolerable level.

Next I'm going to finalize the password cracking part of my project, and then try and solve the problems relating to loading a large file into memory.

Week 4

During this week, I've been mostly programming my own implementation of a hash table, and the program doesn't use any of Java's built-in data structure classes anymore. As I mentioned before, I had already programmed a crude version of a hash set. This implementation, however, required a lot of reprogramming. For instance the hash set version didn't support values for keys, and the set was not iterable (meaning that one could not simply loop through the entries in the set/map). When I finally got the hash table implementation done, I noticed that it was about three times as fast as Java's built in HashMap-class. This is probably due to the fact that the hash table doesn't have to be rehashed, since the approximate size is already known when it is created.

I also tested my program's performance with NetBeans' built-in profiler, and learned that most of the most time-consuming process in hash cracking was reading the table file from disk. The second most time-consuming process was calculating the MD5-hashes needed for chain generation. This is good news for me, since I now know that my implementation of hash table is working somewhat efficiently. When it comes to the table generation process, well, I started creating a table for passwords of length 5-12 using alphanumeric charset on an Ukko-cluster 24 hours ago, and it hasn't completed even 5% of the task yet... I would really like to use multithreading for this issue, or maybe do the table generation on a GPU, but I'm not sure if I have time to implement such things.

Other than that, I did some heavy refactoring on classes MD5Crack and TableCreator, and also moved all printing to UIHelper class. In addition to that, I revised the tests for each class and wrote some new ones for the newly created hash table. I also updated the javadoc.

Currently I am very pleased with my creation, and I think it is ready now, regarding the goals of this course. Next I'm going to ponder how to implement multithreading to table creation, and how to make my program support large table files (meaning files that don't fit in the RAM). I'm also going to look for other parts of the code which I could optimize.