Hair, Hashing and Birthdays

Egan Dunning
Mathematics / Pacific Lutheran University

March 16, 2017

Abstract

This paper will briefly explain the pigeon-hole principle and its connection to the birthday paradox. We will also examine an algorithm to find hash function collisions using the birthday paradox.

1 Introduction

We will use the pigeon-hole principle as a starting point to a discussion of hash functions and the birthday paradox. We will focus on hash function collisions and how the birthday paradox applies to finding these collisions.

2 The Pigeon-hole Principle

The pigeon-hole principle states that if we have n pigeon-holes and $n+1$ objects, and we distribute all objects among the pigeon-holes, then there is a pigeon-hole with at least two objects. This principle is true for every distribution of objects. We observe the worst case, that is, an even distribution of the first n objects. So every pigeon-hole has one object. Now we place the last object in a pigeon-hole. This pigeon hole has two objects now.

This principle can be used to solve fun problems such as determining if there exists two women who have the same number of hairs on their head, or if there are two people in a room who have a birthday on the same day.

3 The Birthday Paradox

The birthday paradox is similar to the pigeon-hole principle, and is related to the birthday problem above. The birthday paradox answers the question: How many people do we need in a room in order for the probability of two people in the room to share a birthday to be 1/2?

The Birthday Paradox. *If 23 people are in the same room, there is a one in two probability that two people share the same birthday.*

Proof. Let A be the event that 2 people in the room share the same birthday. Let $A^c = 1 - A$ be the event that everyone in the room has a unique birthday. If there is one person p_1 in the room, $P(A_1^c) = 1$. If another person p_2 comes into the room, the probability that p_2 shares a birthday with someone in the room is $P(A_2^c) = 364/365$. If person p_3 enters the room, $P(A_3^c) = 363/365$. We repeat this process for all 23 people. Since events $A_1^c, ..., A_2 3^c$ are independent,

$$P(A^c) = 365/365 * 364/365 * ... = \prod_{i=0}^{23} \frac{365 - i}{365}$$

We find $P(A) = 1 - P(A^c) = 0.507297234324$ numerically.

□

Note that $23 \approx 1.2 * \sqrt{365}$. We can now generalize the birthday paradox.

Generalization of the Birthday Paradox. *If we randomly place* $1.2 * 2^{n/2}$ *items in* 2^n *pigeon-holes, the probability that some pigeon-hole contains 2 items is* $1/2$.

An application of this paradox is useful in computer security.

4 Hashing

Hash functions are used in computer security to help determine the authenticity of messages and data and to verify passwords. A hash function maps an input of arbitrary size onto a fixed size output, and should not be easily invertible. We measure the size of the output in bits.

A concrete example of an application of hash functions are checksums. A checksum is a way to check if two copies of the same file have any differences. Often websites that host file downloads will also post the file's checksum, so the user can check that their copy of the file produces the same checksum. The way this works is as follows: the file's owner computes the hash of the file, called the checksum. The person who downloads the file computes the hash of the downloaded file. If the checksums match, the downloaded file is correct. If the checksums don't match, the downloaded file can't be trusted.

A cryptographic hash function satisfies the following properties:

- Pre-image resistance. Given h, finding m such that $hash(m) = h$ is difficult.
- Second pre-image resistance. Given m_1, finding m_2 such that $hash(m_1) = hash(m_2)$ is difficult.
- Collision resistance. Finding unique m_1, m_2 such that $hash(m_1) = hash(m_2)$ is difficult.

We won't look at the first two properties of cryptographic hash functions. But we can apply the birthday paradox to find hash function collisions. We can interpret a hash function with n-bit output size (also called digest size) as a machine that puts inputs into 2^n pigeon-holes. So, if we wanted to use the pigeon-hole principle to find a collision, we would need to try $2^n + 1$ unique inputs to guarantee that two inputs are in the same pigeon-hole.

Since popular hash functions have a digest size ranging from 128 to 512 bits, this method would take a very long time! However, we can reduce the amount of time to find a collision by applying the birthday paradox.

5 Implementation

We can write a program to test the correctness of the birthday paradox. The algorithm to find hash collisions is simple:

1. Generate $1.2 * 2^{n/2}$ unique strings $m_1, m_2, ..., m_k$.
2. Compute the hash of each m_i, $Hash(m_i) = h_i$
3. For each hash pair h_1, h_2 check $h_1 = h_2$.
4. If $h_1 = h_2$, output h_1, h_2 and corresponding m_1, m_2 where $Hash(m_1) = h_1$.
5. If no collision was found, go to step 1.

I used this algorithm to find collisions for the `hashCode()` function included in the java programming language. This function is simple and outputs a java integer, which is a 32 bit value. Java's `hashCode()` is not a cryptographic hash function, but we will use this function as an example, since finding collisions for a cryptographic hash function such as SHA1 takes much more computing power, due to the large (128-bit plus) digest size. In comparison, finding a collision for java's `hashCode()` can be done on inexpensive computers in very little time, making it ideal for this demonstration.

The collision-finding algorithm is mostly trivial. The hardest part is generating unique random strings. We have to decide which characters to use and how long each string should be. In my implementation, I randomized the length of each string, within a range of 20 to 30 characters. Each character in the string is chosen randomly from a set of characters including digits, uppercase letters, lowercase letters and some special characters. This set has 74 elements. Because of the high variability between each string, we can assume each string is unique. This saves some computation time because we do not need to check for uniqueness.

After all the strings are generated, we calculate the hash of each value using `hashCode()`. We store the strings and their hash values in separate arrays.

Now we iterate through every value in the array of hashes and look for a collision. If we use a simple nested loop, we will do extra work, since each pair

of hash values will be compared twice. To avoid this extra computation, we can use the following loop structure:

```
n = 1.2 * 2^(digest/2)
for i = 0 to n do
  for j = i + 1 to n do
    if hash[i] = hash[j] do ...
```

This loop will test equality for each pair of hash values (excluding pair h_1, h_2) exactly once, which is ideal.

Something we might be interested in is: how many iterations of the algorithm must be run in order to find a hash collision? The birthday paradox states that we have a 1 in 2 chance of finding a collision for every $1.2 * 2^{n/2}$ unique random strings that we test. So, on average, the algorithm should iterate twice to find a collision. We can test the average number of iterations to find a collision by running our algorithm many times. After finding 10,000 collisions, the average number of iterations to find a collision was 1.9513. This is close to 2, so the birthday paradox is accurate in this case.

6 Next Steps

One way to find hash function collisions is by using the pigeon hole principle - for each possible hash value, generate a unique string, then compare the hash values of all of these strings. We have seen an algorithm that can find hash function collisions more efficiently using the birthday paradox. However, many parameters in our algorithm can be tweaked.

Will we get a performance increase if we shrink the number of random strings? This would make each iteration of the algorithm faster, since fewer comparisons are made per iteration. However, the algorithm will have to repeat more times, since the chance of finding a collision during one iteration will decrease. To find the optimal number of random strings n, we could plot n versus running time.

Another parameter we can tweak is the string length, and what set of characters we sample from. Do we find collisions faster with a shorter string length? What about if our strings only contain letters? A more interesting investigation could use data encapsulated in a file instead of random strings. For example, generate PDF files, and try to find two PDF files that map to the same hash value.

There are many things to experiment with. My code (link below) can be used as a starting point to answering these questions.

7 Application

Another interesting, and more practical, experiment would be to take some fixed data, for instance a python script, then find another meaningful python script

that maps to the same hash value. For example, we could compute the hash of the fixed file, then generate files with this content:

```
print "Hello world"
#[random string]
```

If we can find a collision between the fixed file and the above file, then both files have the same checksum. So we can have a trusted file that actually shouldn't be trusted! Suppose instead of just printing "hello world", the above file downloads a virus when the unsuspecting user runs the script! I modified the code I wrote to find random hash function collisions to do this.

8 Conclusions

The birthday paradox is an interesting extension of the pigeon-hole principle. Both can be used to find hash function collisions. Here, we used the birthday paradox to find hash function collisions more efficiently than we can using the pigeon-hole principle.

Hash functions are used to verify passwords and files downloaded from the internet. A cryptographic hash function must be collision-resistant, among other things. We used the `hashCode()` function, which is included in the java programming language, in a demonstration. We found that the birthday paradox does apply to finding hash function collisions, as expected.

References

[1] *Mathematical Vistas*, Hilton, Holton, Pederson 2002

[2] *Cryptography I*, Dan Boneh, https://www.coursera.org/learn/crypto

My code is here: https://github.com/egandunning/birthday-paradox
Includes programs for: finding the product in the birthday paradox proof, finding hash collisions for the `hashCode` function, finding how many rounds of the hash collision algorithm requires to find many different collisions, and generating a collision between two specific python files.