Julian Reed
CS109 - Chris Piech
12/2/24

How Good is this Hash Function?

Introduction and Background

In the vast world of data structures and runtime analysis, one approach reigns above them all: hashing. From my early days in CS106B, it was made abundantly clear that when approaching any problem, we should first see if either a hashtable or hashmap could be used before examining any other options. Hashing is even used in other disciplines, like operating systems and cryptography, to quickly encode values. In order to perform hashing, one needs to select a hash function, which will encode an input (int, string, etc.). Hash functions can vary widely based on their purpose. Thus, it is hard to evaluate, in the general case, what is a good hash function and a bad hash function. However, most people agree that there are at least four key attributes to a good hash function (based on the four key attributes identified by Sean Szumlanski in his CS106B lecture):

1. A hash function should be deterministic.
2. Given a uniform set of inputs, a hash function should produce a uniform set of hash codes.
3. The function should produce a large range of values to avoid collisions.
4. The function should produce very different hash codes for similar inputs.

If we are given a specific hash function, can we say how good it is? That is the question that my project seeks to answer.

Methods and Analysis

There are many factors that contribute to forming a good hash function, and quantifying exactly how good a hash function is can be entirely different depending on the use case and other parts of the function's purpose. To evaluate the quality of a given hash function, I measured its performance according to the four key attributes of a "good" hash function described above.

Metric 1: A hash function should be deterministic

Because this can be difficult to test in code, along with the fact that it is generally known whether or not a given function is deterministic, I did not use this metric to evaluate the given hash function. My code assumes that whatever hash function it is given will be deterministic (though all calculations will be mathematically accurate regardless of the function's determinism).

Metric 2: Given a uniform set of inputs, a hash function should produce a uniform set of hash codes.

To quantify the uniformity of the given function, I used KL divergence to find how surprised we would be if the user's hash function was observed to be uniform. Specifically, I used the last line of the equation's derivation (as seen below), where Y is the uniform distribution and X is the distribution of the user's hash function.

$$\begin{aligned}
\mathrm{KL}(X, Y) &= \sum_{x \in X} \mathrm{ExcessSurprise}(x) \cdot P(X = x) \\
&= \sum_{x \in X} \Big[\mathrm{Surprise}_X(x) - \mathrm{Surprise}_Y(x) \Big] \cdot P(X = x) \\
&= \sum_{x \in X} \Big[\log_2 \frac{1}{P(Y = x)} - \log_2 \frac{1}{P(X = x)} \Big] \cdot P(X = x) \\
&= \sum_{x \in X} -\log_2 P(Y = x) + \log_2 P(X = x) \cdot P(X = x) \\
&= \sum_{x \in X} \log_2 \frac{P(X = x)}{P(Y = x)} \cdot P(X = x)
\end{aligned}$$

In my code, this meant I first needed to construct the PMF for the hash function, which I did by hashing incrementally larger values through the user's hash function, modding this hash value by the size of the hash table (which I chose to be equal to the desired number of inputs), and incrementing the count at each location every time a value was hashed there. Once I had the PMF of the user's hash function, I iterated through each value in this PMF. Every time, I incremented the total divergence by the log of the current value of the PMF divided by the probability of seeing this value from the uniform distribution, then multiplying this log by the current value of the PMF. I repeated this process for each value in the PMF, eventually returning the total divergence, which represented the distance between the two distributions.

Metric 3: The function should produce a large range of values to avoid collisions.

To evaluate performance along this metric, I simply counted the total number of collisions that occurred. Because my approach assumed separate chaining, I simply incremented the number of instances of the current hash value instead of searching for an empty spot in the table to place the hashed value. If I used linear probing instead, the number of collisions would be higher since there would be fewer and fewer empty spots available as more and more values were hashed.

Metric 4: The function should produce very different hash codes for similar inputs.

To evaluate performance along this metric, I decided that I would generate inputs that only differ from each other by one character, and determine the expected difference in their hash values. To

do this, I (assisted by ChatGPT) wrote a function to generate a string that would rotate a-z, adding a character when the end of the sequence is reached. For example, this function started with ‘a’ as the first string, and would continue as ‘b’, ‘c’, ‘d’, …, ‘aa’, ‘ab’, ‘ac’, …, ‘ba’, ‘bb’, … and so on. In this way, I ensured that the input strings from the hash function would be similar. I then calculated the difference between the hashed value for the current input and the hashed value from the previous input. I summed this difference for each value that I hashed, then divided by the total number of values hashed minus 1 to reach the mean, which is also the final expected value in this case. The mean and expected value are the same because each difference has the sample probability of occurring: $\frac{1}{num\ values\ hashed - 1}$. Summing each individual difference, we see that:

$$\sum_{i=0}^{num\ values\ hashed - 1} |hash(i) - hash(i-1)| \cdot \frac{1}{num\ values\ hashed - 1} = \frac{\sum_{i=0}^{num\ values\ hashed - 1} |hash(i) - hash(i-1)|}{num\ values\ hashed - 1}$$

where the left hand side of the equation is derived from the expected value formula.

How does the code work?

First, the user should input the hash function they want to evaluate, and the number of values they plan to hash. From there, the program analyzes three hash functions as baselines for comparison (SHA256 as the state of the art hash function, SDBM as a good, but not amazing hash function, and a basic hash function that is pretty terrible), followed by analyzing the user’s hash function. To analyze each function, the program builds a PMF for the hash function (by hashing strings generated by the function mentioned above and incrementing the count at each location every time a string is hashed there, before normalizing the entire table), calculates this distribution’s distance from the uniform via KL divergence, records the number of collisions that occurred, and calculates the expected difference between inputs (as described above). After each function has been analyzed, the statistics for the user function are printed alongside how many times larger they are in comparison to the three baseline functions, followed by a final dump of the statistics for each function.

Link to project source code.

Future Directions

This project has many avenues for future expansion. One of the primary future directions for this project is constructing a UI, which will allow the user to neatly input their hash function without the need to edit the project's source code. Also, a UI implementation would allow for cleaner handling of user input, which might include the desired number of values to hash or any specifications on the types of inputs that would be hashed.

This project could also be improved by using some bootstrapping to determine which approach is better for the hash function - separate chaining or linear probing. Both approaches are used in practice, but in order to truly compare the functions, it would be good to evaluate the function according to its most optimal collision resolution strategy.

Finally, and perhaps most importantly, it would be helpful to score the hash function's effectiveness on a scale of 1-100. At first this was my plan, but I realized that in order to do this accurately, more data would be required about what makes a good hash function and how the statistics for a "good" hash function compare with those of a "bad" hash function. This task is made especially difficult by the order of magnitudes which values can differ, as evidenced by the disparity in each of the measurements between SHA256 and my basic hash function.

References:

- https://docs.python.org/3/library/hashlib.html
- https://web.stanford.edu/class/archive/cs/cs106b/cs106b.1244/lectures/23-hashing/
- https://web.stanford.edu/class/cs109/lectures/20-InformationTheory/
- https://chrispiech.github.io/probabilityForComputerScientists/en/part4/information_theory/
- https://chrispiech.github.io/probabilityForComputerScientists/en/part4/divergence/

For my project, I used ChatGPT in a variety of ways. First, and perhaps most significantly, I used ChatGPT to evaluate how feasible my project ideas are/were by examining existing datasets. My idea to do this project actually came from asking ChatGPT how Shazam works. It told me that Shazam uses hashing as part of their total algorithm to compare songs quickly. After reading this, I spent some time thinking about hashing and realized that entropy might be a helpful lens to view the efficacy of a hash function through, forming the roots of my project. I also used ChatGPT to write certain parts of my code: a mediocre hash function, to implement the SDBM algorithm in python (and identifying that SDBM might be a good benchmark), and to generate the strings I would be hashing. For more details on where and how I used ChatGPT in my code, see this project's GitHub Repo.