CS1321 Homework 9 Hash Function*

Jeffrey D. Oldham

2000 Apr 25

1 Executive Summary

In CS1321-2 today, Travis Pinney queried how to check that one's hash function implementation is correct. While trying to answer his question, I discovered the hashing scheme did not work as expected.

I still recommend implementing the hash function scheme presented in class:

1. Write a recursive function converting a string into a real number in the range $[0, 1)$. This works by converting each character to its ASCII representation, multiplying by the golden ratio $\phi = (\sqrt{5} - 1)/2$, dropping the digits to the left of the decimal point, and adding the fractional product of 256 and the recursive call.
2. Multiply the resulting fraction by M, truncating the digits after the decimal point.

Since a computer represents real numbers using only a finite number of digits, the answer will differ from the mathematically correct answer, but the scheme is so similar to creating random numbers, which has a similar property of adding, multiplying, and then dropping digits that I think the scheme will still work well.

2 Executive Summary on How to Check the Hash Function

As written in the "Suggestions for Improvement" section of the homework, first implement the `hashTable` class using a hash function always yielding zero. After testing the code for correctness, implement the more sophisticated hash function. To check it, have it print the values it yields. If you wish to more than visually inspect the results, use the histogram program to group together and count the values.

3 Explanation

Warning: The following presentation is not mathematically rigorous, but I hope it presents the main ideas in an understandable manner.

The hash function presented in class conceptually consists of the following parts:

1. Converting the string to a (backwards) base-256 number.
2. Multiplying by the irrational number ϕ and dropping all digits to the left of the decimal point.
3. Multiplying by the vector's size M and truncating.

Knuth writes that this scheme is mathematically sound and performs well in practice [Knu98b, pp. 517–519].

Why does this scheme work? Suppose we are hashing some string $c_0c_1c_2c_3c_4$, which is converted to a base-2 number $C_0C_1C_2C_3C_4$, i.e., $C_0 * 2^0 + C_1 * 2^1 + C_2 * 2^2 + C_3 * 2^3 + C_4 * 2^4$. (We use base 2, not base-256, only to make the notation shorter.) Consider multiplying by an irrational number $0.b_0b_1b_2b_3b_4\ldots$, where b_i represent bits. For the case that $C_0 = \ldots = C_4 = 1$, this is the sum

$$\begin{array}{r} 0.b_0b_1b_2b_3b_4\ldots+ \\ b_0.b_1b_2b_3b_4b_5\ldots+ \\ b_0b_1.b_2b_3b_4b_5b_6\ldots+ \\ b_0b_1b_2.b_3b_4b_5b_6b_7\ldots+ \\ b_0b_1b_2b_3.b_4b_5b_6b_7b_8\ldots \end{array}$$

*©2000 Jeffrey D. Oldham (`oldham@cs.trinity.edu`). All rights reserved. This document may not be redistributed in any form without the express permission of the author.

Since the digits of the irrational number have no repeating pattern, they appear random, and the sum appears random.

If we implement the scheme, converting a string to a base-256 number can yield a number much too large to store inside an `integer` or `double`. Instead, we merge the first two steps to ensure the numbers do not grow too large. Distributing the multiplication by ϕ causes no problems, i.e.,

$$C_0\phi * 2^0 + C_1\phi * 2^1 + C_2\phi * 2^2 + C_3\phi * 2^3 + C_4\phi * 2^4,$$

but the numbers are still too large. Instead we can rearrange the computation to use Horner's rule [Knu98a, Section 4.6.4] to rearrange the computation that permits us to distribute taking the fractional portion.

$$C_0\phi + 2 * (C_1\phi + 2 * (C_2\phi + 2 * (C_3\phi + 2 * (C_4\phi + 2 * (0))))).$$

Let $\mathrm{frac}(x)$ represent the fractional portion of the number x. Mathematically, we can show that

$$\mathrm{frac}(a + b) = \mathrm{frac}(\mathrm{frac}(a) + \mathrm{frac}(b))$$

that is, computing the fractional portions of the operands does not change the answer. Thus,

$$\begin{aligned}&\mathrm{frac}(C_0\phi + 2 * (C_1\phi + 2 * (C_2\phi + 2 * (C_3\phi + 2 * (C_4\phi + 2 * (0)))))) = \\ &\quad \mathrm{frac}(\mathrm{frac}(C_0\phi) + \mathrm{frac}(2 * (\mathrm{frac}(C_1\phi) + \mathrm{frac}(2 * (\mathrm{frac}(C_2\phi) + \mathrm{frac}(2 * (\mathrm{frac}(C_3\phi) + \mathrm{frac}(2 * (\mathrm{frac}(C_4\phi) + \mathrm{frac}(2 * (0)))))))))))),\end{aligned}$$

i.e., we compute the fractional portion of each significant operation.

Surprisingly, when we program this scheme, the resulting answer differs from the correct mathematical answer. Why? The problem is the number of digits in a floating point number that a computer can store is limited. In effect, our computer adds something like

$$\begin{array}{r} 0.b_0b_1b_2b_3b_4+ \\ b_0.b_1b_2b_3b_4+ \\ b_0b_1.b_2b_3b_4+ \\ b_0b_1b_2.b_3b_4+ \\ b_0b_1b_2b_3.b_4. \end{array}$$

Each number contributes a finite number of digits so the answer differs from the mathematical answer that involved summing an infinite number of digits.

Despite this *precision problem*, I claim the scheme is still a good scheme because it closely models how random numbers are generated in computers. Random numbers are generated by multiplying the previous random number by a constant, adding another constant, and then dropping all but the last 31 bits. Thus, I still recommend the scheme.

References

[Knu98a] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, third edition, 1998.

[Knu98b] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, second edition, 1998.